INSTITUTE OF INFORMATION TECHNOLOGY

# Research Project on A Textual-based Technique for Smell Detection


## Author,

Sharafat Ahmed

BSSE0617

## Supervised by,

Md. Nurul Ahad Tawhid

Assistant Professor,

Institute of Information Technology,

University of Dhaka


**Course:** SE – 801

**Date:** December 21, 2017

# Abstract

Code smell detection is very important. A tool was built from the knowledge of this paper. Using textual information, this tool can detect different types of code smells such as Blob, Long Method, Feature Envy, Misplaced Class and Promiscuous Package which are of different nature and different granularity. Code smell detection helps to attain better maintainability because it helps us focus more on refactoring. Textual information can be used to detect code smells which means we can make more generalized tool to detect code smells in more and more programming languages.

# Acknowledgement

Foremost, I would like to express my sincere gratitude and appreciation to Md. Nurul Ahad Tawhid for his guidance and support. He motivated me to throughout this work. He guided me with the things that were unknown to me as I have not done any work of this sort before. I find myself very fortunate to work under his supervision.

I would also like to thank our faculty members specially Dr. Kazi Muheymin-Us-Sakib for guidance and support. I would like to thank staffs of IIT for their cooperation.

I would like to thank my family members for their blessings and support which lifted my spirit. I would also like to thank my batch mates for their support throughout this process.

Last but not the least, I would like to thank almighty GOD for giving me the strength to complete this.

# Table of Contents

# Table of Figures

# 1. Introduction

Creating maintainable software is very important as software industry has to support software after delivery. Code smells are stumbling blocks to creating maintainable software. Code smells are usually an indication to deeper problem in the system [4]. Most code smells can be identified easily by human. But, automated code smell detection is not very accurate yet.

Developers are more concerned about just delivering work rather than upholding standard coding style [33]. So, code smells are introduced in code. Detecting and refactoring code smells are very important to make more maintainable software. Most of the researchers has referred code smell to be a structural characteristic of software. But we have failed to outline a border for most types of code smells. So, we have to try to find a different but more accurate smell detection technique.

There has been many work in the field of code smell detection. Most of the researchers have considered structural characteristics to find code smells. Some suggested ad-hoc manual inspection rules, some also proposed tools to visualize and refactor smells. Code smells can be formulated as optimization problem where search algorithms and genetic algorithms are used. Most previous work relies on metrics like size, complexity etc. Different smells requires different types of metrics. In this work, textual based technique to detect different types of code smells will be used.

## 1.1 Motivation

- **Improving Code Quality:** In software engineering, bugs and change of requirements are eminent. If the code quality is not up to the mark, it takes a lot of time and effort to make changes. Code smell is a big reason for hindrance of code quality.

- **Minimization of Code Smells:** Code smells cannot be eradicated complete [34] as it is a recurring problem. But, understanding the relation of textual information may help us to find more important code smells.
- **Language Independent Way of Detecting Code Smell:** There are many code smell detection tools available. Most of those use metric based approach and those approaches are language dependent. Using textual information for code smell detection may lead to language independent code smell detection tool.

## 1.2 Research Question

To detect code smells, textual analysis of code will be used. Existing tools like JDeodorant, Stench Blossom etc. use structural characteristic of code which have low precision and recall rate. The textual data which lies in the code should be considered. So, this leads to following research question:

- How to find code smells in software?

To detect code smells textual data which can be found in code should be considered. This question is associated with some sub-questions. They are:

1. How textual data will be generated from code?

Following steps will be adopted in order to answering the question:

- Textual elements like source code identifiers, comments will be extracted.
- Information retrieval (IR) normalization process will be run.
- Then normalized data will be weighted using term frequency.

2. How code smells will be found from these data?

Following steps will be adopted in order to answering the question:

- Normalized data will be analyzed.
- Different heuristics will be used to detect different code smells.
- Five types of code smells will be used at first. Those are: Long method, Feature envy, Blob, Promiscuous package and Misplaced class.

## 1.3 Contribution

**Smell-Detector:** A java web project was built to detect five types of code smells which has been described in this study. These types are Blob, Long Method, Feature Envy, Misplaced Class and Promiscuous Package. This tool takes a java project as zip file and then analyses the classes and methods to find out code smells mentioned above.

## 1.4 Organization of the Report

Rest of the report is organized as:

- **Chapter 2**: Here in this chapter, code smell, refactoring and its importance has been described briefly.
- **Chapter 3:** Existing works on code smell, code smell detection, refactoring and maintenance has been discussed in this chapter.
- **Chapter 4:** Proposed code detection technique has been described in this chapter.
- **Chapter 5:** This chapter discusses about the result and how the result answers the research questions of this study.
- **Chapter 6:** This chapter discusses about the threats to the validity of this project.
- **Chapter 7:** Brief summary of whole report has been discussed in this chapter.

# 2. Background Study

Software development is a very challenging job. Developers have to face changing requirements, strict deadlines and also have to uphold coding standards in order to ease the job of maintenance. A regular maintenance and smart development is needed to achieve excellent software quality. Software maintenance is an important part of software engineering where developers resolve faults and bugs and improve software quality and performance. Good design, documentation and testing make maintenance a bit easy [1]. But as requirements always change, it is difficult to maintain original design concept. Also, maintenance work varies from developer to developer. This makes maintenance job tougher [2]. Maintenance cost accounts for almost 90% of total cost in legacy software systems [3]. Adding new features, improving performance, removing bugs and faults, improving code quality are main reasons for these costs. Refactoring is an important part of maintenance job where coding structures are changed without changing external behavior of the software [4]. Software companies give a great emphasis on refactoring job as it is a vital part of software maintenance. It is very difficult to judge software's maintainability as it depends on various factors like developers, tasks and tools etc. [5]. Project deadline plays an important role in software maintainability as developers will likely to be unable to uphold original design concept of product and also coding standard if the timeline is not flexible. And this also leads to incomplete delivery in next phases as there was lack in maintainability. So, companies put on a great deal of emphasis on creating more maintainable software with less effort and time.

## 2.1 Code Smells

### 2.1.1 Description

Code smells are not faults or bugs. These are some indications to deeper lying problems in a system [4]. Code smells are some specific design patterns which make software less maintainable [6]. Testability, understandability, reusability, extensibility are some software traits

which are hit the most by code smells. Some examples of code smells are blob, long method, feature envy, misplaced class etc. Fowler [4] divided a total of 22 code smells in 5 categories: bloaters, change preventers, object orientation abusers, couplers and dispensables.

- **Bloaters:** These are those code smells in which functions, classes, etc. are usually long and it is difficult to understand and work with them. Examples are God Class, Long Method, Data Clumps and Primitive Obsession.
- **Object orientation abusers:** These code smells are result of incorrect or incomplete implementation of Object Oriented Design concept. This category includes temporary field, switch statements and alternative classes with different interfaces.
- **Change preventers:** These are the kind of smells which makes modification of existing software very difficult as modification will demand greater effort. Examples of this category are shotgun surgery, divergent change, and parallel inheritance hierarchies.
- **Couplers:** These types of code smells indicate high coupling or unnecessary delegation. It includes inappropriate intimacy, feature envy, middle man and message chain.
- **Dispensables:** Dispensable smells are the type of smells whose absence would make the code cleaner, more efficient and easier to understand. Examples of this category are comments, duplicate code, dead code, data class, speculative generality and lazy class.

## 2.1.2 Detection Techniques

Researchers have worked on code smell detection for many years. Various types of code smell detection techniques have been proposed over years. Some code smell detection techniques are described below:

- **Manual Technique:** Manual technique of code smell detection requires expertise, knowledge and experience. Manually going through codebase and detecting code smells can be a very tedious job. A reading technique for manual smell detection has been proposed by Travassos et al. [7] where a set of formalized detection rules have been

defined for detecting code smells of Object Oriented codebase. Manual detection technique is reliable for human attachment but, they are time consuming, inefficient and developer oriented.

- **Architecture Based Technique:** Architecture design of a system is very important. Knowledge about architecture design can help developers to find areas where code smells may reside [8]. So, developers could actually detect these smells upfront and slow the process of degradation.
- **Metrics Based Technique:** Metrics based techniques deviates a lot in term of suggesting a design to be good. Marinescu [9] suggested a higher abstraction level when working with metric. He also presented a legitimate means to express and quantify good design in software.
- **Machine Learning Technique:** In machine learning technique, the detector tool is trained with code smell data and based on training, the detector can detect new code smells [10].
- **Mining Version History Technique:** In this technique the main idea is to detect code smells by change history found in version controlling system [11].
- **Textual Technique:** Palomba et al. proposed a technique to find code smells by analysing textual information like comments, identifiers etc. First they created a technique for detecting long methods only [12]. Then they suggested a technique for five types of code smells which are long methods, blob, feature envy, misplaced class, misplaced package [13].

## 2.1.3 Code Smell Detection Tools

There are many tools available like JDeodorant, PMD, Strench Blossom, Décor, Iplasma, Infusion etc. Code smell detection results vary a lot among these tools.

Detecting code smells are very subjective and there is no standard to define code smells. Different tools implements detection techniques differently. Some tools like JDeodorant, PMD,

Strench Blossom etc. are plugin tools which help developers in development time to detect smelly code. Some of the tools like JDeodorant give developers option to refactor the code, too.

## 2.2 Refactoring

Refactoring is improving design of codebase without affecting the external behavior of software. The term refactoring was first coined by William Opdyke in 1992 [14]. With Martin Fowler's "Refactoring: Improve the Design of Existing Code" book, the term become more popular. Martin Fowler gave examples of some java code refactoring in his book which made refactoring more practical, too. In refactoring, the structure, organization of codebase is altered to facilitate good software traits like better maintainability, reusability and extensibility.

William Opdyke [14] said it is easier to do refactoring in object oriented code than other type of codes. It is easier to rearrange object oriented code without changing external behavior and it is also easier to add feature if the code is clean. Refactoring should not be mixed up with rewriting as rewriting is the act which changes external behavior of code and refactoring does not change the external behavior of software.

## 2.3 Refactoring Techniques

Fowler [4] suggested to refactor codes whenever you add a function or whenever you need to fix a bug or whenever you do a code review. In his book, Fowler described and gave examples through some java code to demonstrate refactoring techniques.

Existing refactoring categories are described below:

- **Composing Methods:** The techniques which fall into this category systemize/streamline methods, remove code duplications and assignments to parameters, introduce explaining variables etc. to make way for future improvements. The refactoring techniques which are in this category are extract method, inline temp, replace temp with query, Inline Method, split temporary variable, substitute algorithm, and remove assignments to parameters.
- **Feature Moving between Objects:** These are the techniques which moves features between classes, create method on server for delegate hiding, generate new classes from existing ones and conceal implementation specifics. This helps to prevent public access to information which should not be accessible publicly. This category comprises with techniques like move method, extract class, hide delegate, introduce local extension, remove middle man and move field.
- **Organizing Data:** These techniques assist in data handling, disentangling of class association to make them more reusable and replacing primitive variables with rich class. Examples of this category are self-encapsulate field, change bidirectional association to unidirectional, replace data value with object, replace array with object, duplicate observed data, change unidirectional association to bidirectional, replace magic number with symbolic constant, change reference to value and change value to reference.
- **Simplifying Conditional Expression:** These techniques work with logics which will become more complex later. Examples of this category are decompose conditional, consolidate conditional expression, consolidate duplicate conditional fragments, remove control flag, replace nested conditional with guard clauses, replace conditional with polymorphism, introduce null object and introduce assertion.
- **Simplifying Method Call:** These techniques are used to make method calls simpler and easier to understand. This category consists of techniques like separate query from modifier, parameterize method, replace parameter with explicit methods, preserve whole object, rename method, add parameter, remove parameter, replace parameter with method call, introduce parameter object, remove setting method, hide method, replace constructor with factory method, replace error code with exception and replace exception with test.
- **Dealing with Generalization:** These techniques are used to migrate responsibilities along with hierarchies of class. It introduces new interfaces and classes. Examples of this

category are pull up constructor, pull up field, push down method, push down field, pull up method, extract superclass, extract subclass, extract interface and replace delegation with inheritance.

- **Big Refactoring:** In this category, data records are turned into objects, separate domain logics are turned into individual domain classes and subclasses are introduced when needed. It includes tease apart inheritance, separate domain from presentation, extract hierarchy and convert procedural design to objects.

## 2.4 Effect of Code Smell in Maintainability

Code smells have an impact on software maintainability. Code smells like blob, long methods etc. make code difficult to understand and also make modification tough. Although, researchers has related maintainability with code smell but, there has not been any controlled studies where professional software engineers have participated to find out relationship between software maintainability and every types of code smell. Dhaka et al. [15] showed energy consumption to remove different types of code smells empirically.

## 2.5 Summary

In this chapter, code smells, code smell detection techniques, code smell detection tools have been described. Also, refactoring, refactoring techniques and effect of code smells on software maintainability has been described.

# 3. Literature Review

This chapter comprises of literature related to code smell detection techniques and tools. Later importance of this paper is also stated.

## 3.1 Software Maintainability Metrics

Software metrics are the functions which are used to measure different types of characteristics of software. Software maintainability cannot be directly calculated by software metrics but can be understood different maintenance process attributes like time required for making change [16]. So, metrics actually plays a good role in understanding relationship between code smell and maintainability.

Software metrics are mainly of two kinds. One is dynamic metrics and static metrics.

**Dynamic Metrics:** These metrics are used to understand run-time characteristics like dynamic coupling between objects, dynamic message count etc. of software systems.

**Static Metrics:** These metrics are used to understand static properties or characteristics like line of code, number of methods etc. of software systems.

Brief descriptions of some important static metrics are given below:

- **Lines of Code (LOC):** This metrics expresses the total number of lines of a code without counting blank and comment lines. Maintainability decreases with increase of lines of code.
- **Number of Classes(NOCl):** This metrics indicates total number of classes in the codebase.

- **Cyclomatic Complexity Number (v(G)):** This indicates complexity of software systems. Maintainability is influenced by cyclomatic complexity as maintainability decreases with increase of cyclomatic complexity.
- **Maintainability Index (MI):** It is used to show the maintainability of a system. Tools like Visual Studio provides a scale of zero to hundred to demonstrate maintainability index of software.
- **Relative Logical Complexity (RLC):** This is calculated by dividing number of binary decision of a system with number of statements. Higher RLC indicates less maintainability.

## 3.2 Impact of Code Smell

Code smells have its impact on maintenance in terms of time and cost. Code smells make source code error prone [17]. Studies have proved that classes and methods containing code smell are more change prone and the size of change is higher in smelly code [18]. So, it can be said code smells have big influence on software maintainability and it should be studied more in depth.

**Impact on Maintainability:** According to definition, code smells make code less maintainable, less extensible and less understandable. Bois et al. [19] performed a controlled experiment in 2006, where they decomposed God Classes into a number of small classes by performing refactoring. They found that decomposed classes have a significantly higher understandability than the original God Classes. Yamashita et al. [20] observed a host of industrial java projects and the developers of those projects while they performed various maintenance tasks in 2013. They found out, most of the maintenance problems are caused by single or multiple code smells.

**Impact on Fault-proneness:** Code smells make code less maintainable so the likelihood of introduction of bug is very likely. Fowler [4] claimed in his book that, performing refactoring makes code easier to understand hence enables to identify faults easily. In 2014, Zazworka et al.

[21] studied with four types of technical debt of software. One of those four types of debts was code smell and found its relation with threats.

**Impact on Change-proneness:** As code smells make code more fault-prone, developers have to fix codes more frequently. Olbrich et al. [22] in 2009, studied two open source projects Lucene and Xerces to inspect change frequency and change size. They found significantly more change proneness in files containing God Class and Shortgun Surgery smells.

So, we found that code smells have great impact on software. In an ideal world, developers would remove all the code smells. But, that is not always possible as there is a constant pressure of deadline and also lack of experience in developers. Peter et al. [23] observed seven open source projects and found that developers are less likely to refactor even when they are aware of it. This indicates that in real-life, maintaining smell free code is not always feasible.

## 3.3 Code Smell Detection Techniques

There are many ways to detect code smells. It is very important to detect code smell in order to refactor them. Research related to code smell detection started out in 1999 when Fowler [4] introduced different refactoring techniques to remove code smells.

Many researchers have used different metrics to detect code smells in software. These kinds of techniques use different metrics and aspects of software to locate and detect code smells. They apply a threshold to detect code smell but same threshold cannot be applied to detect all code smells as different smells have different implementation variations. Oliveira et al. [24] proposed relative threshold to detect code smells as metrics usually do not apply for every classes of software.

Visualization based techniques are semi-automated techniques to detect code smells [25]. This technique requires expertise of developer and also it is not a scalable technique. This technique is time consuming and inefficient.

Different strategies are applied to identify smells by different techniques. No technique is considered as the ultimate technique to detect all 22 code smells described by Fowler [4].

## 3.4 Code Smell Detection Tools

Many tools have been built over years to detect and remove code smells. Some tools are automated and some are semi-automated. As manual review and detection of code smells are tedious and inefficient, tools are used widely for this purpose. These tools do not usually detect all types of code smells. They do not also support every language.

A brief description of some code smells detection tools are given below:

- **JDeodorant:** Tsantalis et al. [26] introduced this tool. It is plugin software for eclipse. It detects five types of code smells namely God Class, Feature Envy, Duplicated Code, Long Method and Type Checking. It applies the most effective refactoring technique by ranking refactoring techniques for a particular type of code smell.
- **PMD:** This tool analyzes code to find different code smells like dead code, empty try catch block, switch statements and duplicate code. It detects three types of code smells which are long method, long parameter list and large class.
- **Stench Blossom:** Murphy-Hill et al. [27] introduced stench blossom, a code smell detection technique which shows code smells visually. It is also an eclipse plugin and can detect eight types of code smells.
- **DÉCOR:** Moha et al. [28] presented this tool announced this tool which uses a Domain Specific Language (DSL) using high level abstraction to identify code smells. Four types of code smells like blob, spaghetti code, functional decomposition and Swiss army knife has been specified by DÉCOR.
- **iPlasma:** Marinescu et al. [29] introduced iPlasma in 2005. It detects several disharmonies of code. It detects code smells like Feature Envy, God Class, Duplicated Code and Refused Parent Bequest.

## 3.6 Proposed Textual Based Detection Technique

There are three main steps to detect code smells using textual based technique. They are: i) Textual Content Extractor, ii) Information Retrieval (IR) Normalization Process and iii) Smell Detector.

i. **Textual Content Extractor:** In this step textual information like source code, comments, identifiers are extracted from project. Only information needed are extracted.
ii. **Information Retrieval (IR) Normalization Process:** In this step, five sub steps are applied.
    a. In this step, using the camel case splitting, separation of composed identifiers are done. This splits words based on underscores, capital letters and numerical digits.
    b. Here, extracted words are reduced to lower case letters.
    c. In this step, removal of special characters, programming keywords and common English stop words are done.
    d. Here, words are stemmed to their original roots via Porter's stemmer [30].
    e. Finally, to reduce the relevance of too generic words that are contained in most source components, the normalized words are weighted using the term frequency - inverse document frequency (tf-idf) [31] schema.
iii. **Smell Detector:** Normalized words are then analyzed by smell detector. Latent Semantic Indexing (LSI) [32] is used to determine similarity among words of components and also among components. The similarity values are then used to detect different types of smells.

## 3.5 Textual Based Detection

All these tools need different metrics to detect different code smells. They also need different rules. In my base paper, Palomba et al. [13] introduced textual based techniques for code smell detection. In this technique code smells are detected using textual relationship. Five types of

code smells are detected by this technique which is long method, feature envy, blob, promiscuous package and misplaced class.

## 3.6 Summary

Literature related to different code smell detection techniques and tools are presented in this chapter. Also Impact of code smell on software is demonstrated to show importance of code smell detection. It also discusses my base paper and how Palomba et al. [13] used textual information to detect code smells.

# 4. Smell Detector

This chapter contains procedure that was followed to implement the proposed smell detector tool. This tool was built using Java language and "Spring Boot" framework was used to implement the web application. This web application takes a java project as input and as output, it shows the result of 5 types of code smells which are Blob, Long Method, Feature Envy, Misplaced Class and Promiscuous Package detected in those projects.

## 4.1 Smell Detector Tool

Smell Detector takes a java project as input and based on the textual information of that project it run analysis and find code smells in that project. This tool is built using java and takes java project as input. Process of this smell detection is described in the following subsections.

### 4.1.2 Architecture of Smell Detector

There are three main steps for detecting code smells. They are: i) Textual Content Extractor, ii) IR Normalization Process and iii) Smell Detector [13]

I. **Textual Content Extractor:** The first step is to collect all textual information needed for the analysis. In this step, from the source code files, necessary textual information are extracted from every component.

II. **IR Normalization Process:** using a typical Information Retrieval (IR) normalization process, identifiers and comments of each component are normalized. The following steps are performed [31]:
    a. Separating composed identifiers using the camel case splitting which splits words based on underscores, capital letters and numerical digits
    b. Reducing to lower case letters of extracted words
    c. Removing special characters, programming keywords and common English stop words

d. Stemming words to their original roots via Porter's stemmer [30]. Using the term frequency - inverse document frequency (tfidf) schema [31], the normalized words are weighted to reduce the relevance of too generic words that are contained in most source components.

III. **Smell Detector:** The normalized textual content of each code component is then individually analyzed by the Smell Detector. The smell detector uses Latent Semantic Indexing (LSI) [32] which models code components as vectors of terms occurring in a given software system. LSI uses Singular Value Decomposition (SVD) to cluster code components according to the relationships among words and components. Here each code component is given a probability number and according to that probability, it is decided that if the code component is affected with code smell or not.

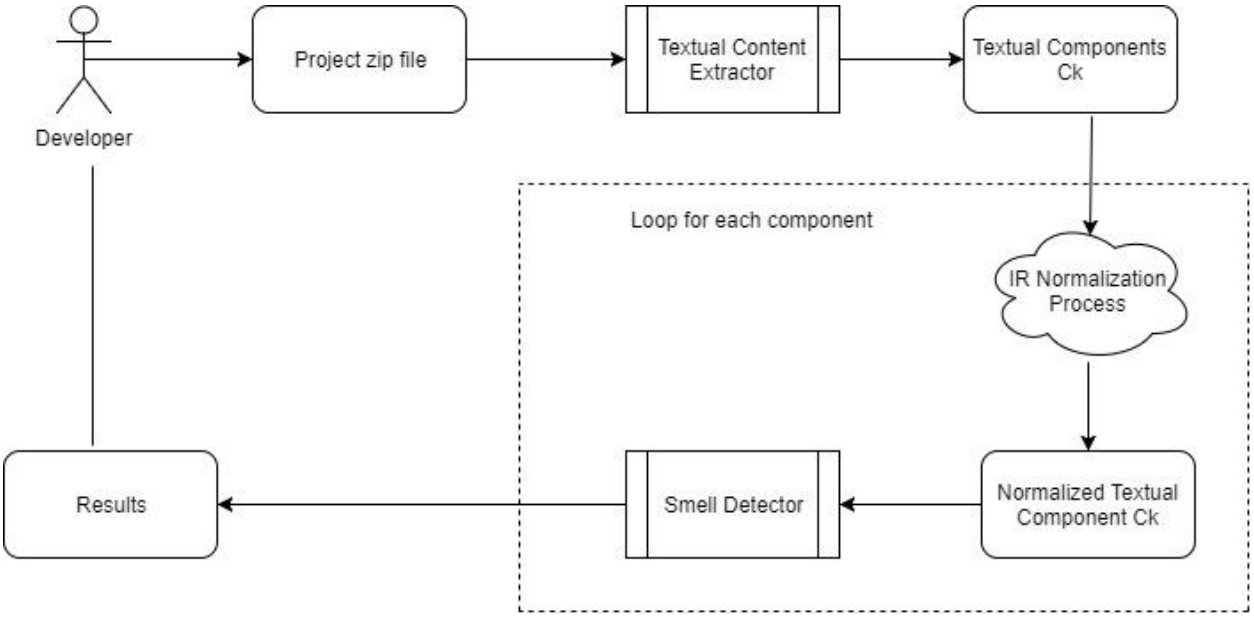


**Figure 1: Proposed Code Smell Detection Process**

## 4.1.2 Code Smell Rules

The rules used to detect code smells in five types of code smells are given below [13]:

- **Long Method:** This smell arises when a method does when a method is responsible for some auxiliary functionality along with its main functionality. The name of the smell does not fully express the smell type. It is more about functionality than size. So, a method can be relatively larger and even then not considered to be a long method. In the proposed tool, consecutive statements are taken into account. Let, $S = \{s_1, s_2, \dots, s_n\}$ be the statements of a method M.

  $$\text{MethodCohesion}(M) = \text{mean}_{i!=j}\ \text{sim}(s_i, s_j)$$

  This is the value of the method's cohesion. The value lies 0 to 1. The lesser the cohesion is, the more chance is there that the method is in fact a long method.

  So, the probability of a method being a long method is,

  $$P_{LM}(M) = 1 - \text{MethodCohesion}(M)$$

  This value also ranges from 0 to 1. The bigger the value is, there is a greater chance of the method to be a long method.

- **Feature Envy:** When a method is more interested in other class rather than its own, it is considered feature envy [4]. So, if a method M is in Class $C_o$ and $C = \{c_1, c_2, \dots, c_n\}$ is the set of classes which shares at least one term with M then, we try to find the most closest class for M using following formula:

  $$C_{closest} = \arg\max_{Ci \in Crelated}\ \text{sim}(M, C_i)$$

  If $C_{closest}$ is not equal to $C_o$, then M shows feature envy.

- **Blob:** Blob are the classes with low cohesion among its methods. Sometimes it is also labeled as god classes. If C is the class under analysis and M = $\{m_1, m_2, \ldots, m_n\}$ is the set of methods of C, then the following rule can be applied to find if C is a blob.

  $\text{ClassCohesion}(C) = \text{mean}_{i!=j}\ \text{sim}(m_i, m_j)$

  This also ranges from 0 to 1. The less the value is, the more chance is there for C to be a blob.

  So, the probability of a class being a blob is

  $P_B(C) = 1 - \text{ClassCohesion}(C)$

  This value lies in 0 to 1. Bigger value indicates a higher chance of a class to be a blob.

- **Promiscuous Package:** If a package contains classes implementing too many features, making it too hard to understand and maintain, it can be considered as promiscuous. Let $C = \{c_1, \ldots, c_n\}$ be the set of classes in P, the textual cohesion of the package P is:

  $\text{PackageCohesion}(P) = \text{mean}_{i!=j}\ \text{sim}(c_i, c_j)$

  The value ranges from 0 to 1 and bigger probability value is better.

  So, the probability of a package to be a promiscuous package is,

  $P_{PP}(P) = 1 - \text{PackageCohesion}(P)$

  This value lies in between 0 and 1. Greater value suggest a package to be a promiscuous package.

- **Misplaced Class:** If a class is not related to the package it is in, then it is called misplaced class. So, if a class C is in Package $P_o$ and $P = \{p_1, p_2, \ldots, p_n\}$ is the set of

packages which shares at least one term with C then, we try to find the most closest package for C using following formula:

$P_{closest} = \arg\max_{Pi \in Prelated} sim(C, P_i)$

If $P_{closest}$ is not equal to $P_o$, then C shows feature envy.

## 4.2 Environmental Setup

The case study was performed in the following setup –

- **Machine:** Intel core i5 second generation processor with clock speed of 2.2 GHz, 4GB of RAM.
- **Operating System:** Microsoft Windows 10 64bit operating system.
- **Java Virtual Machine:** Java runtime environment 8.
- **Bytecode Generator:** Bytecode generated by Eclipes,
- **Version:** Oxygen
- **Source Code:** source code of this study can be found in github.com/sabir001/smell-detection
- **Input Data:** A java project as a zip file.

## 4.3 Summary

This chapter provided the methodologies and rules used to detect code smell using textual information. Also the environment used to build the web application also described here in this chapter.

# 5. Result Analysis

In this chapter, results acquired from tool will be discussed to answer the research question asked in chapter 1. A tool was built as mentioned in previous chapter and various open source projects were analyzed with the tool in order to answer the research question.

Earlier in this report, a research question was asked with two sub questions. They are:

**Research question (RQ):** How to find code smells in software?

**Sub Question 1 (SQ1):** How textual data will be generated from code?

**Sub Question 2 (SQ2):** How code smells will be found from these data?

With the help of the results gathered from the web application described in this report, these questions will be answered in the following sections.

## 5.1 Description of Considered Projects

Six small projects have been considered to be dataset of this study. There were two reasons for choosing those six projects. One is, those are comparatively small projects so manually verifying detected code smells will be a bit easier. And another is, we wanted to test this tool on less popular open source project where some coding standards like variable naming convention are not strictly maintained.

Four of these projects are student projects done by BSSE 6$^{th}$ batch students of Institute of Information Technology, University of Dhaka. Other two projects are sample java spring boot projects. These two projects were taken to reduce internal validity threat of all projects being student projects.

Large scale enterprise level projects were not taken as it would be very difficult to verify all the smells detected by the tool manually. So, calculation of precision of the web application would be very difficult in case of enterprise level big projects.

Brief detail of each project is given below:

**Smell-detection [35]:** It is the smell-detection tool web application built for this report.

**LIPS [36]:** This is a language independent program slicing project.

**Mobile Monkey [37]:** It is a load testing tool built for android.

**RootsPad [38]:** It is an android application.

**GS Spring Boot [39]:** It is a simple spring boot application.

**GS Uploading [40]:** It is also a simple spring boot application for demonstrating uploading.

## 5.2 RQ1: How to find code smells in software?

As mentioned throughout this report, there are many techniques for code smell detection. Manual technique [7] is more accurate but it is not scalable and also it requires expertise. There is architecture based technique [8] and many metrics based techniques [26], [27], [28] and also even machine learning based technique [10].

Textual based code smell detection approach has been applied in the proposed tool. Here in this report, this will be demonstrated why it might be a good idea to consider textual based approach. Also here in this section, the sub questions asked will also be answered.

### 5.2.1 Results of the Projects

The six projects mentioned above was tested by our tool smell-detector. Results of each individual project are described below.

**Smell-detection [35]:** After using smell-detection project as input in our Smell-detector tool, the tool suggested there are seven blobs in this project. After manually checking these seven projects, we saw, these seven classes were indeed blobs.

It also shown there are 65 long methods in this project and after manual inspection, we found that only 16 were indeed long methods. Most of the false positives were resulted in Potters algorithm and LSI calculation classes where identifier names were not proper names. So, textual based approach gave a very bad result in this case.

The tool also showed there was one misplaced class in the project which indeed was a misplaced class.
No result of Feature Envy and Promiscuous Package were recorded.

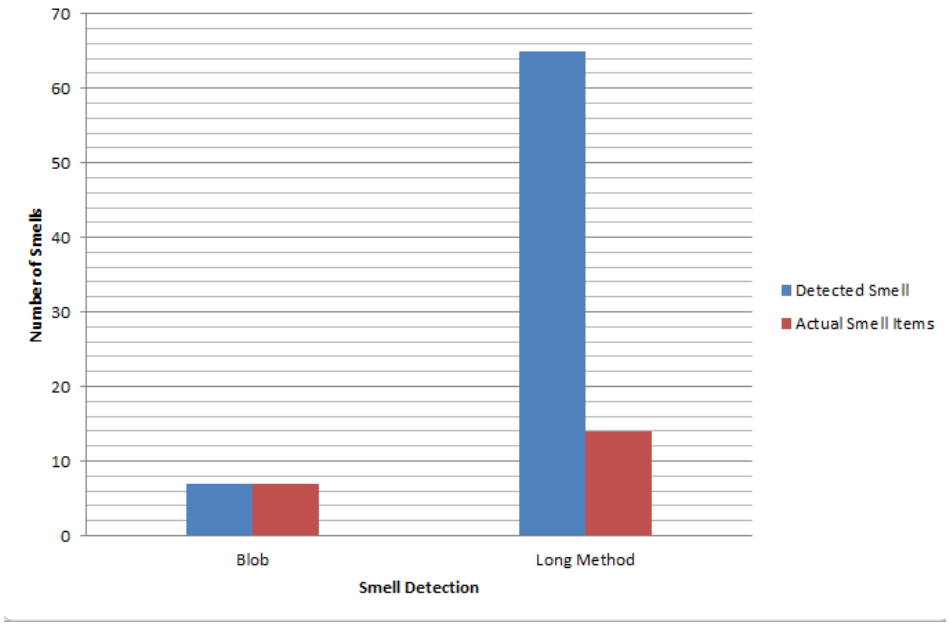


**Figure 2: Results of Smell Detection Project**

**LIPS [36]:** When we used LIPS project as input in our Smell-detector tool, the tool suggested there are four blobs in this project. After manually checking these four projects, we saw, two of them were actually blobs.

This also shown there were 23 long methods among which 11 were actually indeed long methods. No Feature Envy, Misplaced Class or Promiscuous Package was found by the tool in this project.

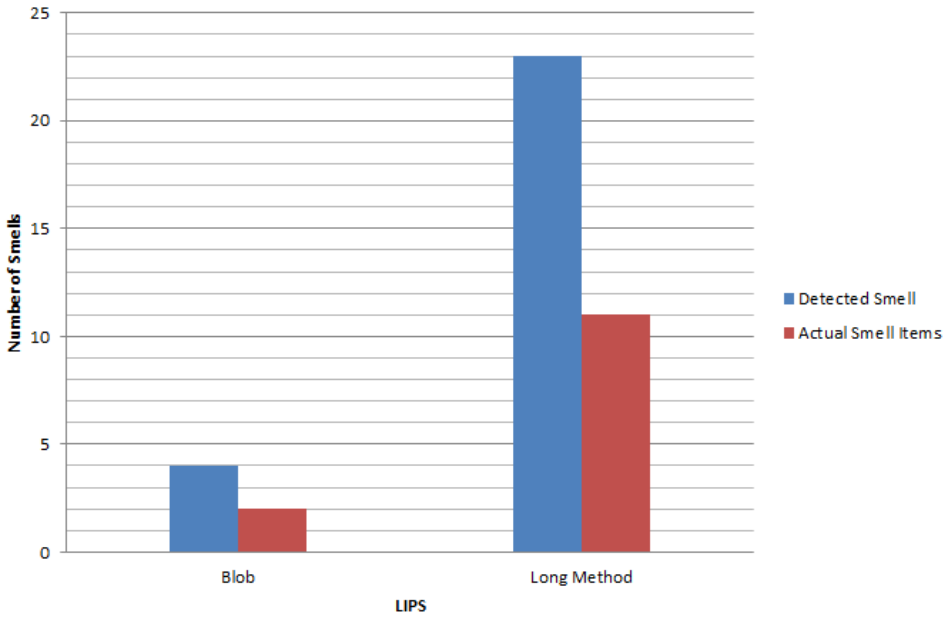


**Figure 3: Results of LIPS Project**

**Mobile Monkey [37]:** The tool suggested there was one blob in Mobile Monkey project which was actually a blob. It also suggested there were 6 long methods in this project. After manual inspection, 3 out of these 6 were found out to be long methods.

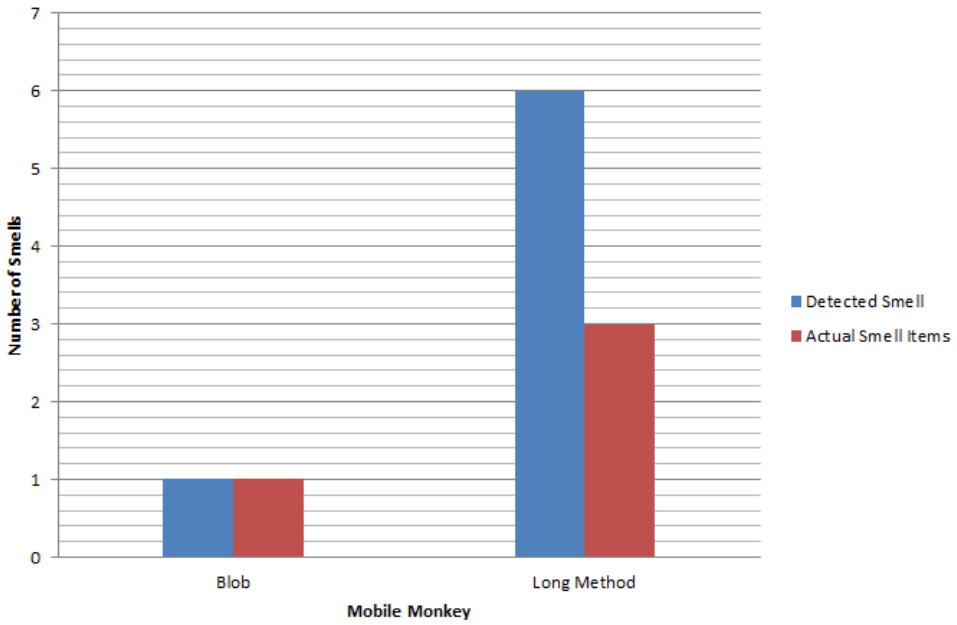


**Figure 4: Results of Mobile Monkey Project**

**RootsPad [38]:** The tool also shown there were 6 blobs in RootsPad project. After manually inspecting these 6 classes, 5 were believed to be blobs. Here precision was very good.

But, the tool suggested 48 methods to be long methods out of which only 30 were found out to be long methods.

For this project, 4 classes were reported as Misplaced Class. Manual inspection suggested, 2 of them were actually Misplaced class.

No result for Feature Envy and Promiscuous Package were recorded.

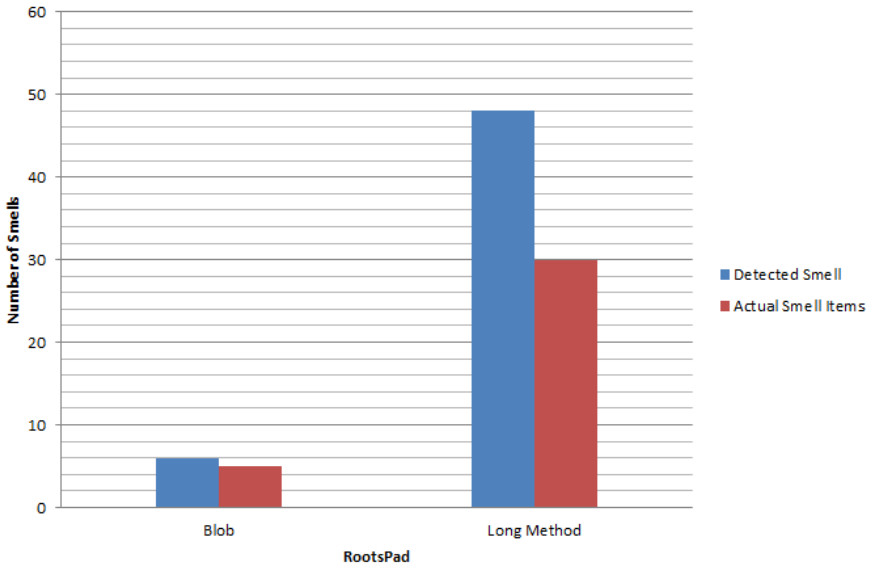


**Figure 5: Results of RootsPad Project**

**GS Spring Boot [39]:** For this project, only one smell was recorded. Only one method was reported as a long method. But, manual inspection suggested it was not actually a long method.

No other type of smell was recorded for this project.

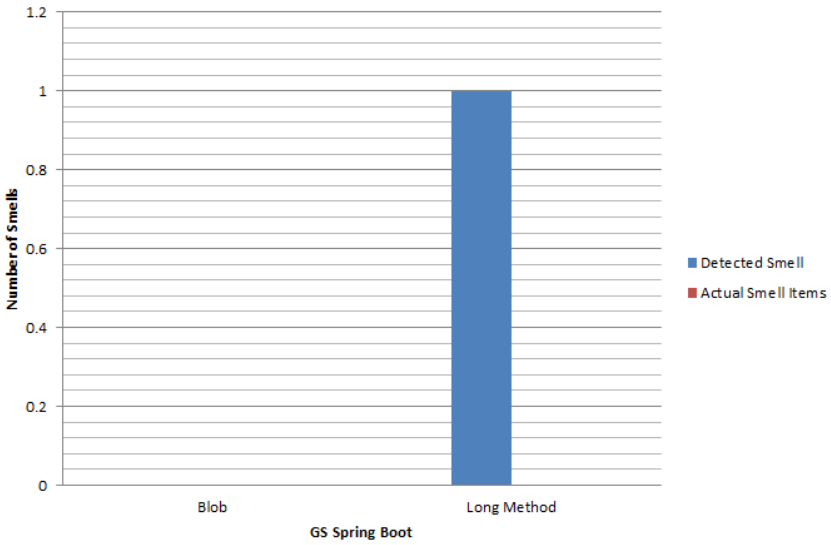


**Figure 6: Results of GS Spring Boot Project**

**GS Uploading [40]:** For this project two classes were reported as blobs by the tool. Manual inspection suggested, both of these are indeed blobs.

The tool also suggested there were 8 long methods in this project. Manual inspection suggested out of these 8 methods, 5 were long methods.

No other type of code smell was reported by the tool.

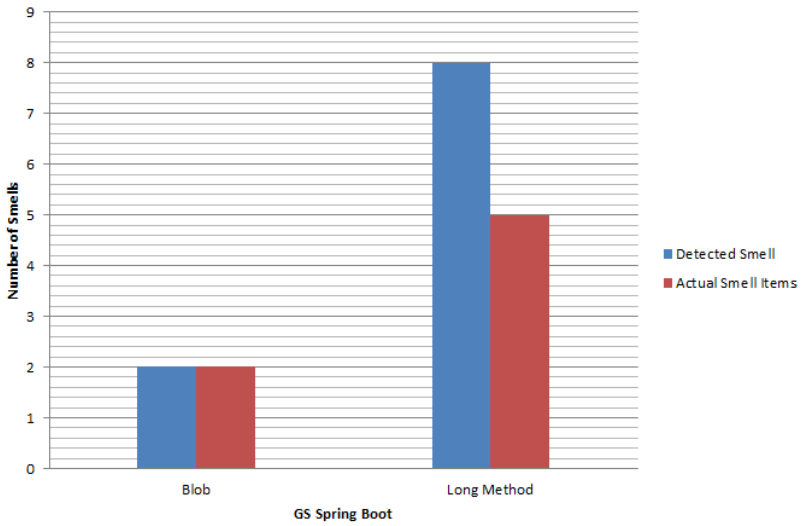


**Figure 7: Results of GS Uploading Project**

## 5.2.2 Result Summary

From the result described above, some conclusions can be drawn. These are described in this section. This data are sole

- Blob has precision of 85% precision
- Long Method has 43% precision
- Misplaced Class has 60% precision
- Proper naming of variables and identifiers results in better precision

- Feature Envy and Promiscuous Package are most rare among the five code smells we are working with

### 5.2.3 Answer to the Research Question

After analyzing the results of the six projects under consideration, we saw we can indeed detect code smells from textual information of a project. So, textual based information can be used to determine code smells. In the chapter 4 of this report, we have discussed about the methodology of extracting textual data. Also discussed how these data will be used to detect code smells.

## 5.3 Summary

Here in this chapter, the results of the projects under consideration were discussed. Some good ideas about textual based code smell detection were also generated from this analysis. Next chapter will conclude this report.

# 6 Threats to Validity

This section describes the threats which can affect the validity of this study.

## 6.1 Construction Validity

Four student projects and two normal projects have been considered for this study. All of those four student projects are from BSSE 6$^{th}$ Batch of Institute of Information Technology, University of Dhaka. This leaves us with construction validity as the result might be biased.

## 6.2 Internal Validity

The cut-off rate or threshold used for this study is a source of threat to internal validity. As the project analyzed by this tool were known to the author of this study, the projects were analyzed multiple times by this tool with different threshold to find a good threshold value. Code smell detection largely depends on this threshold value. So, this is a threat to internal validity. Also understanding the requirements from base paper [13] of this study is also a threat to internal validity.

## 6.3 External Validity

In this study, some code smell types of different level and granularity has been considered. There might also be other types of code smells which can be generated by textual information analysis but was not considered in this study. This possesses a threat to external validity to our study.

# 7. Conclusion and Future Direction

## 7.1 Conclusion

A web based tool called Smell-detector has been developed to find the code smells of software. We found out a precision of 85% for the results of Blobs, 43% for the results of Long Methods and 60% for the results of Misplaced Class. So, the results suggest we can actually detect code smells from textual information of a project.

## 7.2 Future Direction

This work can be much improved in future. Here some direction to the future work is discussed.

In this study, only textual information was used to identify code smells. In future, we can use both metrics based technique and textual based technique to identify code smells. This can give better precision and recall.

Also we can approach towards a more language independent way to detect code smells. If required textual information can be detected from other programming languages, this tool or extension of this tool may find code smells from other languages.

# Reference


1. D. L. Parnas, “Software aging,” in Proc. Int’l Conf on Software Engineering (ICSE). ACM/IEEE, 1994
2. Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, Audris Mockus, “Does Code Decay? Assessing the Evidence from Change Management Data”, in IEEE Transactions on Software Engineering, 2001
3. Erlikh, L., 2000. “Leveraging legacy system dollars for e-business.” IT professional, 2(3),
4. Fowler, M. and Beck, K., 1999 “Refactoring: improving the design of existing code.” Addison-Wesley Professional.
5. Anda, B., 2007, October. “Assessing software system maintainability using structural measures and expert assessments.” In Software Maintenance, 2007. ICSM 2007. IEEE International Conference
6. Olbrich, S.M., Cruzes, D.S. and Sjøberg, D.I., 2010, September. “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
7. Travassos, G., Shull, F., Fredericks, M. and Basili, V.R., 1999, October. “Detecting defects in object-oriented designs: using reading techniques to increase software quality.” In ACM Sigplan Notices
8. Macia, I., Garcia, A., Chavez, C. and von Staa, A., 2013, March. “Enhancing the detection of code anomalies with architecture-sensitive strategies.” In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference
9. Marinescu, R., 2004, September. “Detection strategies: Metrics-based rules for detecting design flaws.” In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference
10. Fontana, F.A., Zanoni, M., Marino, A. and Mantyla, M.V., 2013, September. “Code smell detection: Towards a machine learning-based approach.”InSoftware Maintenance (ICSM), 2013 29th IEEE International Conference

11. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D. and De Lucia, A., 2015. “Mining version histories for detecting code smells.”IEEE Transactions on Software Engineering
12. Palomba, F., 2015, May. “Textual analysis for code smell detection.”InProceedings of the 37th International Conference on Software Engineering-Volume 2
13. Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, Andy Zaidman, “A Textual-based Technique for Smell Detection”, in 2016 IEEE 24th International Conference on Program Comprehension (ICPC),
14. Opdyke, W.F., 1992. “Refactoring object-oriented frameworks.”
15. Dhaka, G. and Singh, P., 2016, December. “An Empirical Investigation into Code Smell Elimination Sequences for Energy Efficient Software.” In Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific
16. Frappier, M., Matwin, S. and Mili, A., 1994. “Software metrics for predicting maintainability.” Software Metrics Study: Tech. Memo
17. M. Vokac, “Defect frequency and design patterns: An empirical study of industrial code,” Software Engineering, IEEE Transactions on, vol. 30, no. 12, pp. 904–917, 2004.
18. F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness,” in Reverse Engineering, 2009. WCRE’09. 16th Working Conference on, pp. 75–84, IEEE, 2009.
19. B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, “Does god class decomposition affect comprehensibility?,” in IASTED Conf. on Software Engineering, pp. 346–355, 2006.
20. A. Yamashita and L. Moonen, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” in Proceedings of the 2013 International Conference on Software Engineering, pp. 682–691, IEEE Press, 2013.
21. N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, et al., “Comparing four approaches for technical debt identification,” Software Quality Journal, vol. 22, no. 3, pp. 403–426, 2014
22. S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, “The evolution and impact of code smells: A case study of two open source systems,” in Proceedings of the 2009 3rd

international symposium on empirical software engineering and measurement, pp. 390–400, IEEE Computer Society, 2009.

23. R. Peters and A. Zaidman, “Evaluating the lifespan of code smells using software repository mining,” in Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, pp. 411–416, IEEE, 2012.
24. for source code metrics,” in Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on, pp. 254–263, IEEE, 2014.
25. Carneiro, G.D.F., Silva, M., Mara, L., Figueiredo, E., Sant'Anna, C., Garcia, A. and Mendonca, M., 2010, September. “Identifying code smells with multiple concern views.” In Software Engineering (SBES), 2010 Brazilian Symposium on (pp. 128- 137). IEEE.
26. Tsantalis, N., Chaikalis, T. and Chatzigeorgiou, A., 2008, April. “JDeodorant: Identification and removal of type-checking bad smells.” In Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on (pp. 329-331). IEEE
27. Murphy-Hill, E. and Black, A.P., 2010, October. “An interactive ambient visualization for code smells.” In Proceedings of the 5th international symposium on Software visualization (pp. 5-14). ACM
28. Moha, N., Gueheneuc, Y.G., Duchien, L. and Le Meur, A.F., 2010. “DECOR: A method for the specification and detection of code and design smells.” IEEE Transactions on Software Engineering, 36(1), pp.20-36.
29. C. Marinescu, R. Marinescu, P.F. Mihancea, D. Rat¸iu, and R. Wettel, “iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design”, in ICSM Industrial and Tool Volume, 2005
30. M. F. Porter, “An algorithm for suffix stripping,” Program, vol. 14, no. 3, pp. 130–137, 1980.
31. R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval. Addison-Wesley, 1999
32. S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by latent semantic analysis,” Journal of the American Society for Information Science, no. 41, pp. 391–407, 1990.

33. Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, " Do they Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," in 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 1
34. A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the, pp. 106–115, IEEE, 2010.
35. "Smell-detection – github repository", https://github.com/Sabir001/smell-detection Accessed: December 20, 2017
36. "LIPS – github repository", https://github.com/tawsif93/LIPS Accessed: December 20, 2017
37. "Mobile-Monkey – github repository", https://github.com/mehedi-iitdu/Mobile-Monkey-Java Accessed: December 20, 2017
38. "Rootspad– github repository", https://github.com/Nasimuzzaman/Roostpad Accessed: December 20, 2017
39. "GS Spring Boot – github repository", https://github.com/spring-guides/gs-spring-boot Accessed: December 20, 2017
40. "GS Uploading – github repository", https://github.com/spring-guides/gs-uploading-files Accessed: December 20, 2017