GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

January, 2018

REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

AUTOMATED AND METRIC-BASED DETECTION OF CODE SMELLS AND ANTIPATTERNS

SAMER AL-RUBAYE

MSc. THESIS
DEPARTMENT OF COMPUTER ENGINEERING
PROGRAM OF COMPUTER ENGINEERING

ADVISOR
ASSIST.PROF. DR. YUNUS EMRE SELÇUK

İSTANBUL, 2018

REPUBLIC OF TURKEY

YILDIZ TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

AUTOMATED AND METRIC-BASED DETECTION OF CODE SMELLS AND ANTIPATTERNS

A thesis submitted by Samer Al-Rubaye in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** is approved by the committee on 09.02.2018 in Department of Computer Engineering.

Thesis Adviser

Assist. Prof. Dr. Yunus Emre SELÇUK

Yıldız Technical University

Approved By the Examining Committee

Assist. Prof. Dr. Yunus Emre SELÇUK

Yıldız Technical University ______________________

Assoc. Prof. Dr. Mehmet S. AKTAŞ, Member

Yıldız Technical University ______________________

Prof. Dr. Selim AKYOKUŞ, Member

Doğuş University ______________________

ACKNOWLEDGEMENTS

I would first to thank my thesis advisor Assist. Prof. Dr. Yunus Emre Selçuk of the Computer Engineering Department at Yıldız Technical University, his door was always open whenever I had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it.

Finally, I must express my very profound gratitude to my parents and my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process or researching and writing this thesis, this accomplishment would not have been possible without them.

January 2018

Samer Al-RUBAYE

TABLE OF CONTENTS

LIST OF SYMBOLS

%	Percentage

LIST OF ABBREVIATIONS

ADP	Acyclic Dependencies Principle
CYC	Cyclic Dependency
LOC	Line of code
OO	Object Oriented

LIST OF FIGURES

Page

LIST OF TABLES

Page

ABSTRACT

AUTOMATED AND METRIC-BASED DETECTION OF CODE SMELLS AND ANTIPATTERNS

Samer AL-RUBAYE

Department of Computer Engineering

MSc. Thesis

Adviser: Assist.Prof. Dr. Yunus Emre SELÇUK

In software engineering, Patterns are techniques which are used to improve the design and enhance the reusability of a solution to commonly occurring problem design and they are general solutions which are used for common problems in object-oriented systems.Antipatterns and code smells are opposites of design patterns. Those are not bugs: They are not technically incorrect coding and they do not currently prevent the program from functioning.Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. They also make software maintenance more costly. The antipattern concept is introduced as poor solutions to solve recurring problems, even though developers think that they practice a design pattern. Code smells, also called also as bad smells, refer to any symptom in the source code of a program that possibly indicates a deeper problem.

Poltergeist antipattern is one of the software development antipatterns. Poltergeists are classes with limited accountability and roles to be active in the system; thus, their efficient life period is quite short. Poltergeists are messy software design, make needless abstractions; they are overmuch complex, hard to know, and hard to look after. Our objective is to propose a metric based approach to determine whether a class is poltergeist or not.

Keywords: Antipatterns, Poltergeist, Object Orientation, Software Metrics, Detection

YILDIZ TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

ÖZET

KOD KUSURLARI VE ANTİ-KALIPLARIN OTOMATİK VE ÖLÇÜT TABANLI TESPİTİ

Samer AL-RUBAYE

Bilgisayar Mühendisliği Anabilim Dalı

Yüksek Lisans Tezi

Tez Danışmanı: Yrd. Doç. Dr. Yunus Emre SELÇUK

Yazılım tasarım kalıpları, nesne yönelimli sistemlerin modellenmesinde yaygın olarak karşılaşılan problemler için sunulmuş, tasarımı iyileştirerek yeniden kullanılabilirliğini arttıran çözüm önerileridir. Karşıt kalıplar Ve kod kusurları ise tasarım kalıplarının zıttı olan önerilerdir. Bunlar yazılım hatası değildir: Programın doğru çalışmasını engellemez ve hatalı kod olarak nitelendirilemezler. Bunun yerine yazılımın geliştirilme sürecini yavaşlatan veya gelecekte hatalar ile karşılaşma olasılığını arttıran tasarım zayıflıklarının işaretçileridirler. Bakım aşamasının maliyetini de arttırırlar. Karşıt kalıplar, geliştiriciler bir tasarım kalıbı uyguladıklarını düşünseler bile, sık karşılaşılan problemler için zayıf çözüm önerileridirler. Kod kusurları ise bir yazılımın kaynak kodunda büyük olasılıkla daha derin sorunlara işaret eden semptomlardır .

Poltergeist, bir yazılım geliştirme karşıt kalıbıdır. Bu kalıbı ortaya koyan sınıfların sistemde sınırlı rolü bulunup örneklerinin yaşam süresi sınırlıdır. Bu karşıt kalıp sisteme gereksiz soyutlamalar, karmaşıklık ve bakım zorluğu getirir. Tez çalışmasının amacı bir sınıfın bir Poltergeist örneği olup olmadığının belirlenebilmesi için yazılım ölçütü tabanlı bir yaklaşım ortaya koymaktır.

Anahtar Kelimeler: Karşıt kalıplar, Poltergeist, Nesneye yönelim, Yazılım ölçütleri

YILDIZ TEKNİK ÜN İVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

CHAPTER 1

INTRODUCTION

Code smells can be defined as the periodical signs of prosaic layout and coding through [1]. Code smells reduce readability, resilience in addition to increase error-proneness [2], [3], [4], Thus code smells have to be anatomized, found out and. In addition to code smells, Moha refers to potential styling smells [5]. Fowler presented 22 code smells and referred to the necessity of refactoring processes [1].

Software projects generally transact with big outputs which consist of a lot of ingredients, their architectures can easily get very intricate and tough to resolve [6]. The strongest techniques to prohibit these untidy and knotted architectures are the employment of design and detection patterns. Design patterns [7] are common and effective techniques utilized for promote the design and backing the preservability reusability and adverse engineering [8]. Abiding to design patterns to get the better ability of grasp and preservability as well [9].

In software engineering Patterns are techniques which are used to improve the design and enhance the reusability of a solution to commonly occurring problem design and they are general solutions which are used for common problems in object-oriented systems [7]. Antipatterns [1] and code smells [10] are opposites of design patterns. Those are not bugs: They are not technically incorrect coding and they do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. They also make software maintenance more costly. The antipattern concept is introduced as poor solutions to solve recurring problems, even though developers think that they practice a design pattern. Code smells, also called also as bad smells, refer to any symptom in the source code of a program that possibly indicates a deeper problem.

The deeper problem hinted by a code smell can be noticed when the code is subjected to a short feedback cycle where it is refactored in small, controlled steps, and the resulting design is examined to see if there are any further code smells that indicate the need of more refactoring. Most bad smells affecting a piece of code are already present since its creation, rather than being introduced later via evolutionary code changes.

Detection strategies are a composed logical condition based on metrics, but one metric alone cannot answer all the questions. The first step in detection is by identifying the symptoms by breaking down the informal rules in a correlated set of symptoms, the next step is to select the metrics that quantify the best of identifying the properties but there are two alternatives by using a well-known metrics from a well-known metrics suite or from a summarized by various authors, or by defining a new metric so the metric will captures exactly one of the symptoms that appear in that design. The third step is to define for each metric we will use the filter that captures the best symptom; the final step is correlating these symptoms.

Detection approaches are manual detection and automated detection. As thesis work, our goal is to generate an automated tool that detects some code smells and/or Antipatterns. As similar works exist in the literature, we will select at least one smell or antipattern that has not been detected before for unique contribution. We will select more smells or Antipatterns, too. We will propose different metrics for their detection if they are already covered in the literature.

Software systems require maintenance to fit with all-time changes of the demands and the circumstances on the other side to design patterns [11], code and design smells are “weak” solutions to have recourse execution and design problems—may slow down their process by making it difficult for software engineers to implement the changes.

An example of a design smell is the Spaghetti Code antipattern, the Spaghetti Code is uncovered by classes that does not contain structure that announces long methods with no parameters. The classes and methods names may propose practical programming. Spaghetti Code does not take advantage of object-oriented techniques, such like several different forms and inheritance, and block their utilization [5].

Antipatterns look alike patterns; they are utilized as fully undoing, however, adverse to patterns they usually supply erroneous and serious solutions to existent troubles [11]. The term antipattern is coined via [9], [13] and the concept was primarily presented through Akroyd in 1996 as a reflation of pattern, which is generally consumed even although it is an incorrect pursuit [14]. Antipatterns are indigent solutions to periodical design troubles. These miserable solutions unfavorable impact expansion and servicing phases through diminishing comprehensibility from the system, diminishing readability of the source code and decreasing the resilience of the software [15]. Code and design smells consist of low-Level or domestic troubles. Moreover, they are a considerable offer of Antipatterns that are more popular design smells [1], [16]. Because the mischievous effects of Antipatterns, they require being neatly detected and rejected.

We introduce an automated and metric-based detection method of the poltergeist antipattern. Our approach examines both metric values and rule establishment. We use theses metrics to calculate the cyclic dependency using Java class and source file directories and create a design quality metric. In addition to the cyclic dependency, we need to detect the short methods we can find by calculating the line of code (LOC) metric by utilizing a metric plug-in for Eclipse.

1.1 Literature Review

1.1.1 Patterns and Antipatterns

Patterns are mechanisms to develop the layout and promote reusability. Design patterns are utilized for popular troubles in object-oriented systems. They are also defined by other researchers as one of the easiest and powerful techniques used to enhance the design, thus enhance the preservability, reusability, and reproduction engineering [8]. Every pattern characterizes an issue that happen repeatedly again in our medium, thus describes the essence of solutions for an existing problem, in a form which you have the ability for utilizing this solution many times over, wanting never acting it the same approach twice [17], [18] They provide us with a very similar characteristic description of a software pattern, as the object and the commands for producing the object.

The antipattern term is also presented as indigent solutions to cope periodic troubles, although developers consider which they try a design pattern [19]. Antipatterns are invisible while using the pattern detection methods. Antipatterns come from design issues, that they are the obverse of design patterns. They were first presented by [14] as a response to the pattern, which is a wrong try that is frequently used [19], [20], [16]. Antipatterns look alike patterns, they seem as a strong solution while they provide an incorrect solution for the problems [21], [22].

1.1.1.1 Types of Antipatterns

The antipatterns can be examined in 3 groups: Software development, software architecture and project management antipatterns:

a. Software development Antipatterns

Software development is a difficult activity, thus it can easily head to astray from the planned structure as determined by architecture, analysis, and design. Development Antipatterns usually occur at class or package level and they can be eliminated by using various official and unofficial refactoring approaches.

b. Software architecture antipatterns

Architecture antipatterns occur on the system-level and enterprise-level of software applications and components. A strong and extensible architecture is a must for the success of software development. Architecture-driven software development promises quality software without antipatterns. Carefully crafted architectures that comply with design patterns and best practices lead to antipattern-free software.

c. Project management antipatterns

Software project management is a complex task requiring many different skills. If process maturity models such as CMMI, ISO/IEC 90003:2004 or PMI is followed, the probability of project management antipatterns' occurrence is eliminated.

1.1.1.2 Poltergeist Antipattern

Poltergeist antipattern introduces needless and not required navigation paths in the method of development, highly impermanent associations of a specified class with another one, occurring of not recognized classes, the event of a limited period of time and short duration classes or classes that occur only to mention other classes through limited time associations. They also have limited responsibilities and function in the system, Poltergeists antipatterns chaos the software designs, by making unwanted abstractions; they are so complex and difficult to maintain.

Akroyd [14] called these classes "Gypsy Wagons" the reason of their appearance for a while in place and they are gone, however, the Gypsy Wagons are created as observer classes that appear only to mention methods of other classes, generally in a preset series. These classes consider as a bad design for three reasons: Firstly for their unnecessarily consummation of resources every time they appear, secondly their addition cause several excessive navigations paths that are redundant, and finally they interrupt the suitable OO design by their needlessly filling the object model.

1.1.2 Code Smell

Code smells are possible to realize as the periodical signs of prosaic layout and coding through [1]. Code smells are related to helpless coding practices which reason for long-term preservability issues and mask bugs [23]. Code smell is a display of poor design and evolution, problems that stays profound in code and decreases the quality of software [19].

1.1.3 Manual and Auto Detection

The manual detection approaches are used in order to avoid false positive detections in suspicious cases. However, these approaches suffer from large time and effort overhead, mostly in the big systems. They also suffer from some oddities, like context-dependence and the fuzzy definitions. The auto-detection approaches were advanced to solve the problems of times and efforts that are desired especially in the big systems. However, they suffer from the uncertainty problems, which they provide quality analysis with an unsorted set of filtered classes with no signal of which one(s) should be investigated first for confirmation and correction.

Travassos et.al [24] offered their method, where the objective is to define design smells using manual survey and perusal mechanism. Their mechanism depends on only manual search and smells are not fixed. Therefore, it is not suitable to use this method on big systems. [25] Suggested a way which counts on the metric-based appraisal to detect design smells and execute it in the IPLASMA tool. This way needs profound information of metrics to detect an antipattern case. Also, changes in threshold scale may cause very different results. Thus located thresholds are critical and do not afford any error. [26] Tried to discover a trade-off between manual and fully automated detection methods. The goal of their method is gaining a mechanism which does not complain from high time and tension exhaustion in big systems while bypassing suspicion problems.
Some other researchers [27], [28], [29] proposed their manual methods which rely on many manual ratings to detect the Antipatterns. [30] And [31] came up with a fully automated method to deny suspicion and a tall list of problems. They used visualization mechanisms to show detection results. Their research ignores analyst view. [5] Produced a DSL-based (domain specific language) path, DECOR, relying on some set of principles which characterize Antipatterns and formed an antipattern - smell classification. They introduced rule cards and technique to transform those rule cards into antipattern detection algorithms. They paid attention to specific several well-known Antipatterns and attempted to detect these Antipatterns with auto-produced algorithms. Survey of [32], [33], and [34] are other automated methods, which were managed to detect Antipatterns.

1.1.4 Rule based detection

Rule characterization is the keystone of general of the antipattern detection models. These basics are manually clarified by analysts and goal to specify the refer that describe smells. They are created as collections of basically quantitative, constitutional and/or lexical offers [35]. Every smell requires its own detection rule and a correct threshold amount that is a so ticklish decision. So, the number potential antipattern statuses may be so major to detect the Antipatterns manually utilizing above rules [11]. [25], [27], [5], [19], [36] implement rule based detection approaches for different and intersecting Antipatterns and code smells.

1.1.5 BBN (Bayesian belief network) based detection

BBN based detection methods impede uncertainty problems and utilize past outcomes to enhance effectiveness [34]. However, this operation has a high cost and demands more time and knowledge [37]. The aforementioned process needs specialized decisions, consequently, many of candidates will be detected as antipattern instances although they are not antipattern instances. BBN models can be used just within their specific context and cannot easily be generalized. [8].

1.2 Objective of the Thesis

Poltergeists are classes with fixed accountability and roles to be active in the system; thus, their efficient life period is quite short. Poltergeists are messy software design, make a needless abstractions; they are overmuch complex, hard to know, and hard to look after. Poltergeist antipattern is one of the software development Antipatterns, and it have several names in addition to its own name, we can mention some of them (Gypsy, Big dolt Controller class and Proliferation of classes).

Poltergeist Antipattern usually occurs in conditions where designers use object orientation but do not adhere to its best practices. In poltergeist Antipattern, no one is able to identify one or more ghost like occurrence of classes that appear only shortly to start some activity in another more lasting class. [14] Called these classes "Gypsy Wagons" as they appear in one day and get away the next day. Gypsy Wagons are fabricated as observer classes that occur only to recall methods from other classes, generally in a predetermined series. They are generally clear because their names are often attached by _manager or _controller.

The Poltergeist Antipattern is generally meant on the part of some beginner architect who does not have enough information about the object-oriented concept. Poltergeist classes' model poor design for several reasons: Firstly, they are needless, so they waste resources every time they "show." Secondly they are inactive because they use several excessive navigation tracks. Thirdly and lastly, they disturb the object-oriented design by unnecessary cluttering the topic model.

The display and results of poltergeist antipattern can be found by searching the following:

- excessive navigation routes,
- passing associations,
- stateless classes,
- temporal, short-term objects and classes,
- single-process classes that occur only to "seed" or "mention" other classes through temporal associations,
- Classes with "control-like" procedures called such as start_process_alpha [16].

One can notice these classes by checking the cyclic dependency between the classes and delegate methods or short methods by depending on a threshold for "good" values. Our objective is to determine whether a class is a poltergeist instance or not by finding those features. To calculate this dependency we need to use JDepend.

JDepend tests the relation between java packages by breaking the Java class and source file index and by generating layout quality metrics for each Java package. JDepend enables us to automatically obtain and monitor package dependencies.

The first metric we could depend on to find the poltergeist antipattern is finding the short methods by depend on (LOC) methods line of code, which is used to calculate the size of code by calculating the program source code. This metric will be calculated by using a plug-in for Eclipse IDE.

Our literature search has not revealed any work on detecting the Poltergeist antipattern although there are works on detecting other Antipatterns and code smells, as mentioned in the relevant chapter.

The second metric called Dependency Finder has a strong querying instrument depends on Perl regular formula. This tool can show how to use many other methods, and maybe all of them, For example, "Show us all calls to constructors of this class." With other measuring tools, we must select each constructor, have the tool creates a useful information for it, saving the results, and gather them later in an XML file to use it in our software. Dependency Finder can calculate closures, that is, follow dependencies and find all reachable from a given start point. This can be helpful to find the package related components together.

1.3 Hypothesis

In this thesis, we submit a rule-based automated antipattern detection system for object oriented software. Considering the Poltergeist antipattern definition given in the previous chapter, we have selected three important indicators of a class being a Poltergeist instance:

- Condition 1: A Poltergeist instance is a part of high cyclic dependency.
- Condition 2: A Poltergeist instance has short method count higher than average.
- Condition 3: A Poltergeist instance does not have any method that is not longer than 2 times of the average method line of code.

Our hypothesis is formed as follows: If both of the two conditions given above is correct for a class, then that class is a Poltergeist instance. We do not define an abstract class as a Poltergeist instance as no objects can be instantiated from those classes and they can have zero method length in form of abstract methods. However, we examine concrete subclasses of abstract classes.

To verify our hypothesis, we will carry out the following steps: First, we will determine the normal values for the given conditions. Our aim is to detect the Poltergeist anti-pattern instances in Java code in a way that discovers the highest possible instances with the least overhead. Therefore, we have focused on the most visible symptoms of the Poltergeist anti-pattern: Classes having a cyclic association and short methods. If there are at least three classes participating in a cyclic association and one of its participants have low average line of code (LOC) value for its methods, that class is detected as a Poltergeist anti-pattern instance candidate. [30] Have determined the appropriate average length of methods as 10 and they determine the threshold for a method considered to be short as 7 for code written in Java. We will use the same threshold part of condition 2 above.

There are many tools available for object introspection. Therefore, we have decided to make use of such tools. We have selected Dependency Finder [30], a free suite of tools for analyzing compiled Java code. We have used its abilities to obtain an XML file that gives information about class associations and to obtain another XML file that gives LOC metrics. We have written code that uses these files as inputs and displays possible Poltergeist anti-pattern instances.

Dependency Finder can only analyze package-level cyclic dependency. Therefore, our code implements an algorithm [38] to find all the class-level cycles.

CHAPTER 2

ANALYSIS OF RELATED METRICS

Recently, serving costs have increasing exceeded to 50% and arrived at 90% of the total costs of software systems [39] To arrive to minimize the cost of maintenance, the investigators have suggested many methods for facility program understanding, and distinguish alteration- and bug prone sections of the source code of software systems. These methods consist of source code metrics like [40], [41]. And heuristics to impose the design of a software system (e.g., [42], [3], [43]) lately, we have begun on analyzing the effect of Antipatterns on the modification-proneness of software units [3]. Antipatterns [16] are “poor” settling to resolve and accomplishment troubles. Comparing to design patterns [11] that are “fine” solutions to periodic design issues. Antipatterns are commonly presented in software systems by provider’s scarcity the sufficient information or experiment in solving a specific problem or having to misuse several design patterns. [44] Described an antipattern as “provide us with a very similar characteristic description of a software pattern, as the object and the commands for producing the object”. Anterior researches like ours [3], backing this characterization by offering those software units, i.e., classes, influenced by Antipatterns are more probable to bear changes than different units.

Software metrics are applied to provide programmers with a feedback about their program. Metrics can be applied as rules to refactoring. They give a path to calculate the process of code through development.

2.1 Metric tools

A build tools is applied to automate repetitive functions during this operation. This could be compiling root code, running software experiments and generating files and documentation for the software deployment.

In this thesis we are going to use some Metrics tools, an open-source Eclipse plug-in, to calculate some metrics on a project. The plug-in should already be installed into Eclipse.

Dependency Finder is a Group of tools for analyzing gathered Java code. At the essence, it is a strong dependency analysis implementation that reproduced dependency graphs and shots them for helpful information. This Dependency Finder can be found in many in many ways to be used, including command-line tools, a Swing-based app implementation, and a group of Ant tasks.it can also calculate the closures, I used this tool to take an XML file form the java source code to examine it.

2.2 Cyclic Dependency and LOC

Method’s Lines of Code (LOC) is one of the metrics we are going to use: That calculates the total lines of code in the selected method. It is only calculated for non-blank and non-comment lines inside a method. If a method’s LOC is over 50 lines, it is proposed that the method is cracked up for reading and maintaining. In class level, if a class has over 750 lines of code, one needs to divide the class [17].

The other metric will be cyclic dependency (CYC): The cyclic dependencies metric is an extra quality metric to estimate the quality of your code. This metric will provide you a calculation of how many class cycles there is. This metric can also be defined in package level. We should avoid having cycles to proceed with the Acyclic Dependencies Principle (ADP) defined by Robert C. Martin declared that no cycles have to be allowed in the package dependency graph. A CYC value of 0 marks that the package being calculated is not connected in any cyclic dependencies with other packages. In this thesis, we will not calculate a CYC value but examine all classes in cycles with the purpose of Poltergeist detection.

Packages engaged in many cycles are harder to preserve. If you make a modification in one cyclic engaged package, it might have an impact on another package that you were not adjusting firstly because it is engaged in a cycle with the package you were adjusting it. So packages with cycles in the dependency architecture should be re-factored because neither package can be used independently of the other [45].

CHAPTER 3

EXPERIMENTAL RESULTS

It was not an easy task to find some code that contains cyclic class associations, the most visible symptom of Poltergeist anti-pattern. To the best of our knowledge, software in Qualities Corpus Index does not contain any cycles. We were not able to find cycles in code written by undergraduate computer engineering students of our department as their graduation projects, either. However, an old version of the Lucene software [44] used in some graduation projects is found to have cyclic class associations. Lucene 3.0.3 contains multiple cyclic association cases, all occurring in package org.apache.lucene.index. The complete listing of cycles is as follows:

1. #1 DocFieldProcessor → StoredFieldsWriter → StoredFieldsWriter$PerDoc → DocumentsWriter → DocFieldProcessor

2. DocFieldProcessor → StoredFieldsWriter → StoredFieldsWriter$PerDoc → DocumentsWriter → DocumentsWriter$PerDocBuffer → DocumentsWriter → DocFieldProcessor (extension of #1)

3. #3 DocumentsWriter → IndexWriter → IndexFileDeleter → DocumentsWriter

4. DocumentsWriter → IndexWriter → MergePolicy → IndexWriter → IndexFileDeleter → DocumentsWriter (extension of #3)

5. DocumentsWriter → IndexWriter → IndexWriter$ReaderPool → IndexWriter → IndexFileDeleter → DocumentsWriter (another extension of #3) #6 InvertedDocConsumer → DocInverterPerThread → DocInverter → InvertedDocConsumer

6. InvertedDocConsumer → DocInverterPerThread → InvertedDocEndConsumerPerThread → DocInverterPerField → DocInverterPerThread → DocInverter → InvertedDocConsumer (extension of #6)

When the classes that are part of the cycles examined and method lengths are considered, DocFieldProcessor and DocInverterPerThread have been detected as Poltergeist instances. Important information about classes mentioned in the cycles is as follows:

- DocFieldProcessor is the most obvious Poltergeist instance, one can read "This class doesn't do any real work of its own: it just forwards the fields to a DocFieldConsumer." in its class documentation.

- DocInverterPerThread confirms to the Poltergeist description. It only pairs some objects and their consumers, actual work is done between those pairs.

- IndexWriter$ReaderPool confirms to our method's metrics but it is not actually a Poltergeist instance. It manages a pool that includes shared objects. This is a responsibility solid enough to make this class a non-poltergeist.

Metric calculations and evaluations of classes that are in a cycle in Lucene 3.0.3 is given in Appendix A. The resulting confusion matrix is given in Table 3.1 and its interpretation is given in Table 3.2

Table 3.1 Confusion Matrix of Evaluations of Lucene 3.0.3

Lucene 3.0.3 Confusion Matrix		Actual Case	
		Is Poltergeist	Not Poltergeist
Our Approach	Is Poltergeist	2	2
	Not Poltergeist	0	4

Table 3.2 Interpretation of Evaluations of Lucene 3.0.3

Accuracy	Fall-out	Recall	Specificity	Miss Rate
75%	33%	100%	67%	0%

The terms given in Table 3.2 can be explained as follows: Accuracy is a straightforward term indicating how correctly the detection is made. Fall-out means false positive rate, where the estimation is positive but the actual result is negative. Recall, also called as

Sensitivity, means true positive rate, where the estimation is positive and the actual result is also positive. Specificity means true negative rate, where the estimation is negative and the actual result is also negative. For accuracy, recall and specificity, higher value is better where for fall-out and miss rate, lower value is better.

Lucene 3.3.0 contains multiple cyclic association cases, all occurring in packages org.apache.lucene.index and org.apache.lucene.util.fst. In the list below, the first 5 cycles are in the util.fst package and the rest are in the index package:

1- Builder → NodeHash → FST→ Builder$UnCompiledNode → Builder
2- Builder$UnCompiledNode → Builder → NodeHash → FST → Builder$UnCompiledNode
3- FST → Builder$UnCompiledNode → Builder → NodeHash → FST
4- NodeHash → FST → Builder$UnCompiledNode → Builder → NodeHash
5- NodeHash → FST → FST$BytesWriter → FST → Builder$UnCompiledNode → Builder → NodeHash
6- IndexWriter → DocumentsWriter → DocumentsWriter$PerDocBuffer → DocumentsWriter → IndexWriter$FlushControl → IndexWriter
7- IndexWriter → DocumentsWriter → DocumentsWriter$ByteBlockAllocator → DocumentsWriter → IndexWriter$FlushControl → IndexWriter
8- IndexWriter → DocumentsWriter → DocumentsWriter$DocState → DocumentsWriter → IndexWriter$FlushControl → index.IndexWriter
9- IndexWriter → DocumentsWriter → DocumentsWriterThreadState → DocumentsWriter → IndexWriter$FlushControl → IndexWriter
10- IndexWriter → DocumentsWriter → DocumentsWriter$WaitQueue → DocumentsWriter → IndexWriter$FlushControl → IndexWriter
11- IndexWriter → DocumentsWriter → DocumentsWriterThreadState → DocumentsWriter$DocState → DocumentsWriter → IndexWriter$FlushControl → IndexWriter
12- InvertedDocConsumer → DocInverterPerThread → InvertedDocEndConsumerPerThread → DocInverterPerField → DocInverterPerThread → DocInverter → InvertedDocConsumer.

When the classes that are part of the cycles and their method lengths are examined, DocInverterPerThread, Documentswritrer$PerDocBuffer, DocumentsWriter&DocState

and DocumentsWriter$ByteBlockAllocator have been detected as Poltergeist instances. Metric calculations and evaluations of classes that are in a cycle in Lucene 3.3.0 is given in Appendix B. The resulting confusion matrix is given in Table 3.3 and its interpretation is given in Table 3.4

Table 3.3 Confusion Matrix of Evaluations of Lucene 3.3.0

Lucene 3.3.0 Confusion Matrix		Actual Case	
		Is Poltergeist	Not Poltergeist
Our Approach	Is Poltergeist	3	2
	Not Poltergeist	0	2

Table 3.4 Interpretation of Evaluations of Lucene 3.0.3

Accuracy	Fall-out	Recall	Specificity	Miss Rate
71%	50%	100%	50%	0%

The terms given in Table 3.4 can be explained as follows: Accuracy is a straightforward term indicating how correctly the detection is made. Fall-out means false positive rate, where the estimation is positive but the actual result is negative. Recall, also called as Sensitivity, means true positive rate, where the estimation is positive and the actual result is also positive. Specificity means true negative rate, where the estimation is negative and the actual result is also negative. For accuracy, recall and specificity, higher value is better where for fall-out and miss rate, lower value is better.

Lucene 7.0.1 contains multiple cyclic association cases, all occurring in package org.apache.lucene.index. The complete listing of cycles is as follows:

- DocumentsWriter → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter
- DocumentsWriter → DocumentsWriterPerThreadPool → DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter
- DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy →

DocumentsWriterFlushControl → DocumentsWriter → DocumentsWriterPerThreadPool → DocumentsWriterPerThread

- DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → LiveIndexWriterConfig → DocumentsWriterPerThreadPool → DocumentsWriterPerThread
- DocumentsWriterPerThreadPool → DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → DocumentsWriterPerThreadPool
- FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → DocumentsWriterPerThreadPool → DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy
- LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → DocumentsWriterPerThreadPool →DocumentsWriterPerThread →LiveIndexWriterConfig
- LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → LiveIndexWriterConfig
- DocumentsWriter → LiveIndexWriterConfig → DocumentsWriterPerThreadPool → DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter
- DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter →

LiveIndexWriterConfig → DocumentsWriterPerThreadPool → DocumentsWriterPerThread

- FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → DocumentsWriterFlushControl → DocumentsWriterPerThreadPool → DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy
- LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → DocumentsWriterFlushControl → DocumentsWriterPerThreadPool → DocumentsWriterPerThread → LiveIndexWriterConfig
- LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → DocumentsWriterPerThreadPool → DocumentsWriterPerThread →LiveIndexWriterConfig
- LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriterPerThreadPool → DocumentsWriterPerThread → LiveIndexWriterConfig
- DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → LiveIndexWriterConfig → DocumentsWriterPerThreadPool → DocumentsWriterPerThread
- DocumentsWriterPerThread → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriter → LiveIndexWriterConfig → FlushPolicy → DocumentsWriterFlushControl → DocumentsWriterPerThreadPool → DocumentsWriterPerThread

When the classes that are part of the cycles examined and method lengths are considered, DocumentsWriterPerThreadPool have been detected as not Poltergeist.

Metric calculations and evaluations of classes that are in a cycle in Lucene 7.0.1 is given in Appendix C. The resulting confusion matrix is given in Table 3.5 and its interpretation is given in Table 3.6

Table 3.5 Confusion Matrix of Evaluations of Lucene 7.0.1

Lucene 7.0.1 Confusion Matrix		Actual Case	
		Is Poltergeist	Not Poltergeist
Our Approach	Is Poltergeist	0	0
	Not Poltergeist	0	1

Table 3.6 Interpretation of Evaluations of Lucene 7.0.1

Accuracy	Fall-out	Recall	Specificity	Miss Rate
100%	0	---	100%	---

We did not detected any poltergeist in this version. The resulting confusion matrix of all of the tested versions is given in Table 3.7 and its interpretation is given in Table 3.8

Table 3.7 Confusion Matrix of Evaluations of all versions of tested Lucene

Confusion Matrix for Tested Lucene Versions		Actual Case	
		Is Poltergeist	Not Poltergeist
Our Approach	Is Poltergeist	5	4
	Not Poltergeist	0	7

Table 3.8 Interpretation of Evaluations of all versions of tested Lucene

Accuracy	Fall-out	Recall	Specificity	Miss Rate
75%	36%	100%	64%	0

CHAPTER 4

DISCUSSION

The experimental results encourage us to enhance our method. We will extend our experiments to other software.

The interpretation tables in Chapter 3 show that the accuracy of our approach is acceptable and the recall is perfect. However, we can improve our approach in terms of fall-out.

The most important missing feature of our method is to take dependency relationships into account. If a class A has a member of type B, an association relationship from class A to class B occurs. If class A does not have any members of type B but uses instances of class B either as a method parameter or as a temporary variable, a dependency relationship from class A to class B occurs. Our method currently examines only association relationships where work of [45] takes both association and dependency relationships into account.

REFERENCES

[1] Fowler, M. and Beck, K.,(1999). "Refactoring: improving the design of existing code". Addison-Wesley Professional.

[2] Abbes, M., Khomh, F., Gueheneuc, Y.G. and Antoniol, G.,(2011), "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension" . Software maintenance and reengineering (CSMR), 2011 15th European conference on 181-190. IEEE.

[3] Khomh, F., Di Penta, M., Guéhéneuc, Y.G. and Antoniol, G.,(2012). "An exploratory study of the impact of antipatterns on class change-and fault-proneness". Empirical Software Engineering, 17(3):243-275.

[4] Khomh, F., Di Penta, M. and Gueheneuc, Y.G.,(2009), October. "An exploratory study of the impact of code smells on software change-proneness. Reverse Engineering" ,(2009). WCRE'09. 16th Working Conference on 75-84. IEEE.

[5] Moha, N., Gueheneuc, Y.G., Duchien, L. and Le Meur, A.F.,(2010). DECOR: "Amethod for the specification and detection of code and design smells".IEEE Transactions on Software Engineering, 36(1):20-36.

[6] Gamma, E., et.al.,(1994). "Design Patterns – Elements of Reusable OO Software". Addison-Wesley, USA.

[7] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis, S.T.,(2006). "Design pattern detection using similarity scoring". IEEE transactions on software engineering, 32(11).

[8] Din, J., AL-Badareen, A.B. and Jusoh, Y.Y.,(2012), December. "Antipatterns detection approaches in object-oriented design".A literature review. Computing and Convergence Technology (ICCCT),(2012) 7th International Conference on 926-931. IEEE.

[9] Fontana , F.A., Maggioni, S. and Raibulet , C.,(2011). "Understanding the relevance of micro-structures for design patterns detection". Journal of Systems and Software, 84(12):2334-2347.

[10] Fowler, M. ,(1999). "Refactoring : improving the design of existing code". Addison-Wesley, USA.

[11] Vlissides, J., Helm, R., Johnson, R. and Gamma, E.,(1995). “Design patterns: Elements of reusable object-oriented software. Reading”. Addison-Wesley, 49(120):11.

[12] Koenig, A.,(1995). “Patterns and antipatterns”, Journal of Object-Oriented Programming 8 (1):46-48.

[13] Koenig, A.,(1998). “Patterns and antipatterns. The patterns handbook: techniques, strategies, and applications”, 13:383.

[14] Akroyd, M., (1996). “Anti patterns session notes. Object World”.

[15] Okutan, A. and Yıldız, O.T.,(2014). “Software defect prediction using Bayesian networks. Empirical Software Engineering”, 19(1):154-181.

[16] Brown, W.H., Malveau, R.C., McCormick, H.W. and Mowbray, T.J., (1998). “AntiPatterns: refactoring software, architectures, and projects in crisis”. John Wiley & Sons, Inc..

[17] Aleksandra Tešanovic´ Linköping University Department of Computer and Information Science Linköping, Sweden http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.1162&rep=rep1&type=pdf May 2017.

[18] Coplien, J.O.,(1998). “Software design patterns: Common questions and answers. The Patterns Handbook: Techniques, Strategies, and Applications”, 311-320.

[19] Aras, M.T. and Selçuk , Y.E.,(2016). “Metric and rule based automated detection of Antipatterns in object-oriented software systems”. Computer Science and Information Technology (CSIT),(2016)7th International Conference on 1-6. IEEE, Amman, Jordan.

[20] Joydip Kanjilal.Microsoft architect, www.infoworld.com/article.html ,May 2016

[21] Ang, J., Cherbakov, L. and Ibrahim, M.,(2005). SOA Antipatterns. IBM Corp., Nov.

[22] Long, J.,(2001). “Software reuse Antipatterns”. ACM SIGSOFT Software Engineering Notes, 26(4):68-76.

[23] Rogers, J. and Pheatt, C.,(2009). “Integrating antipatterns into the computer science curriculum”. Journal of Computing Sciences in Colleges, 24(5):183-189.

[24] Mannan, U.A., Ahmed, I., Almurshed, R.A.M., Dig, D. and Jensen, C.,(2016), May. “Understanding code smells in Android applications”. Proceedings of the International Workshop on Mobile Software Engineering and Systems 225-234. ACM.

[25] 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 34(10):47-56. ACM.

[26] Marinescu, R.,(2004), September. “Detection strategies: Metrics-based rules for detecting design flaws. In Software Maintenance”,(2004). Proceedings. 20th IEEE International Conference 350-359. IEEE.

[27] Dhambri , K., Sahraoui , H. and Poulin, P.,(2008), April . “Visual detection of design anomalies. Software Maintenance and Reengineering”,(2008). CSMR(2008). 12th European Conference 279-283. IEEE.

[28] Munro, M.J.,(2005). “Product metrics for automatic identification of bad smell" design problems in java source-code. Software Metrics,(2005). 11th IEEE International Symposium 15-15. IEEE.

[29] Ali Kacem, H. and Sahraoui, H.,(2006). “Détection d’anomalies utilisant un langage de description de règle de qualité”, actes du 12e colloque LMO. LMO, Ed.

[30] Ciupke, O., (1999). “Automatic detection of design problems in object-oriented reengineering”. Technology of Object-Oriented Languages and Systems,(1999) . TOOLS 30 Proceedings 18-32. IEEE.

[31] Lanza, M. and Marinescu, R., (2006). “Object-Oriented Metrics in Practice – Using Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems”.

[32] Van Emden, E. and Moonen, L.,(2002). “Java quality assurance by detecting code smells”. Reverse Engineering,(2002). Proceedings. Ninth Working Conference 97-106. IEEE.

[33] Sousa, P. and Ebert, J. eds., (2001). Proceedings of the Fifth European Conference on Software Maintenance and Reengineering. IEEE.

[34] Rao, A.A. and Reddy, K.N., (2007). “Detecting bad smells in object oriented design using design change propagation probability matrix 1”.

[35] Khomh, F., Vaucher, S., Guéhéneuc, Y.G. and Sahraoui, H., (2011). “BDTEX: A GQM-based Bayesian approach for the detection of antipatterns”. Journal of Systems and Software, 84(4):559-572.

[36] Kessentini, M., Sahraoui, H., Boukadoum, M. and Wimmer, M., (2011). “Search-based design defects detection by example”. International Conference on Fundamental Approaches to Software Engineering 401-415. Springer Berlin Heidelberg.

[37] Dependency finder http://depfind.sourceforge.net, May 2017.

[38] Erlikh , L., (2000). “Leveraging legacy system dollars for e-business. IT professional”,2(3):17-23 .

[39] Romano, D. and Pinzger, M., (2011). “Using source code metrics to predict change-prone java interfaces”. Software Maintenance (ICSM), (2011) 27th IEEE International Conference 303-312. IEEE.

[40] Mauczka, A., Grechenig, T. and Bernhart, M., (2009). “Predicting code change by using static metrics. In Software Engineering Research, Management and Applications”,(2009). SERA'09. 7th ACIS International Conference 64-71. IEEE.

[41] Posnett, D., Bird, C. and Dévanbu, P., (2011). “An empirical study on the influence of pattern roles on change-proneness. Empirical Software Engineering”,16(3):396-423.

[42] Thummalapenta, S., Cerulo, L., Aversano, L. and Di Penta, M., (2010). “An empirical study on the maintenance of source code clones. Empirical Software Engineering”,15(1):1-34.

[43] Coplien, J.O. and Harrison, N. B. , (2005) . “Organizational patterns of agile software development. Pearson Prentice Hall”.

[44] Apache lucene core https://lucene.apache.org/core, May 2017.

[45] Stoianov, A. and Şora, I., (2010) “Detecting Patterns and Antipatterns in Software using Prolog Rules”, IEEE Int’l. Joint Conf. on Computational Cybernetics and Technical Informatics (ICCC-CONTI), 27-29 May 2010, Timisora, Romania.

APPENDIX-A

LUCENE 3.0.3

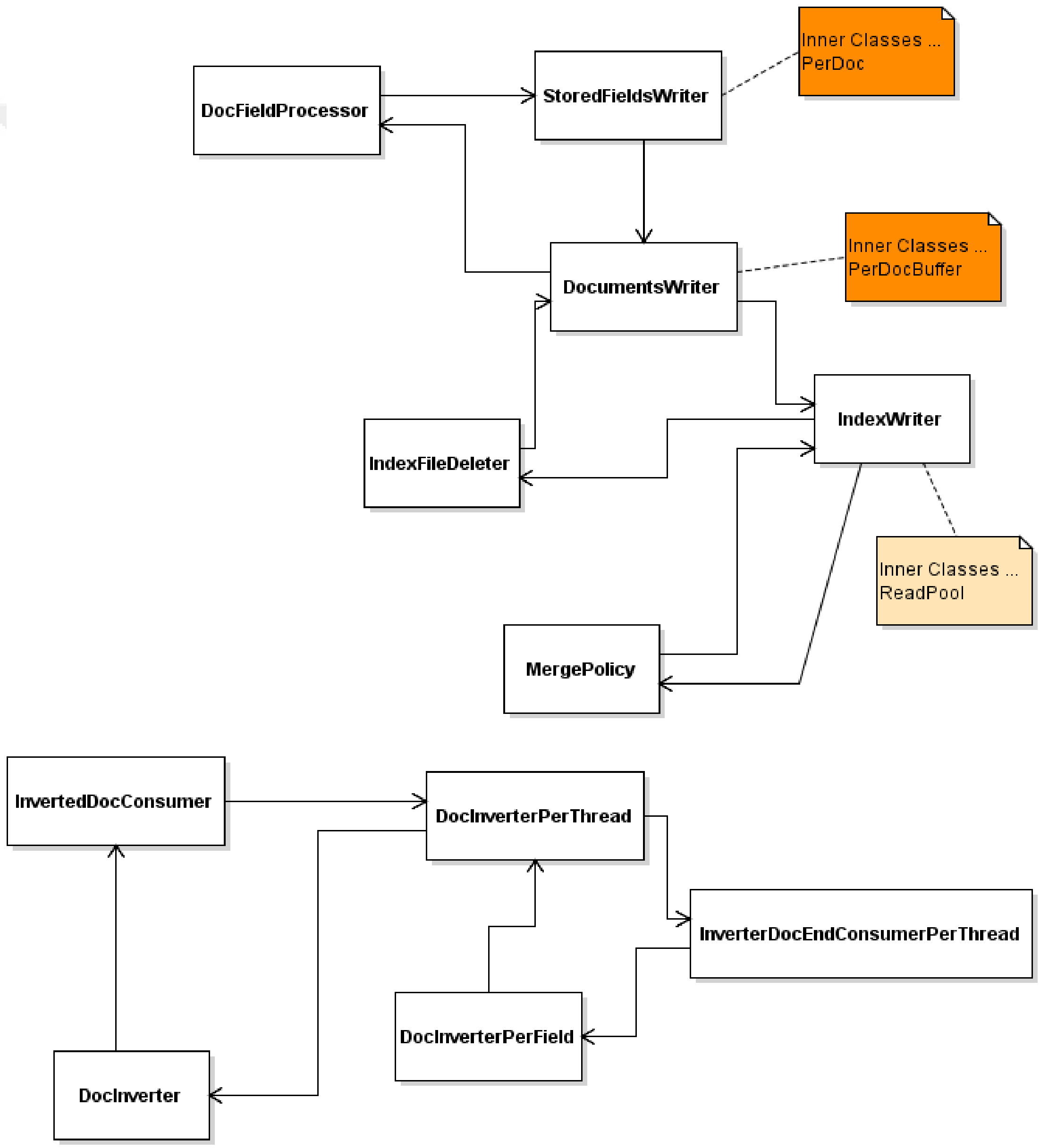

Figure A.1 Lucene 3.0.3 Cycles

APPENDIX-B

LUCENE 3.3.0

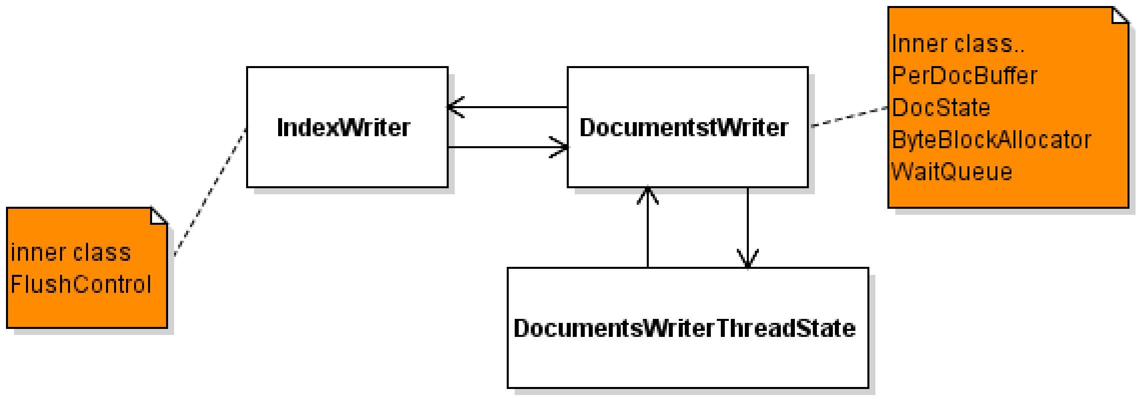

Figure B.1 Lucene 3.3.0 Cycle 1

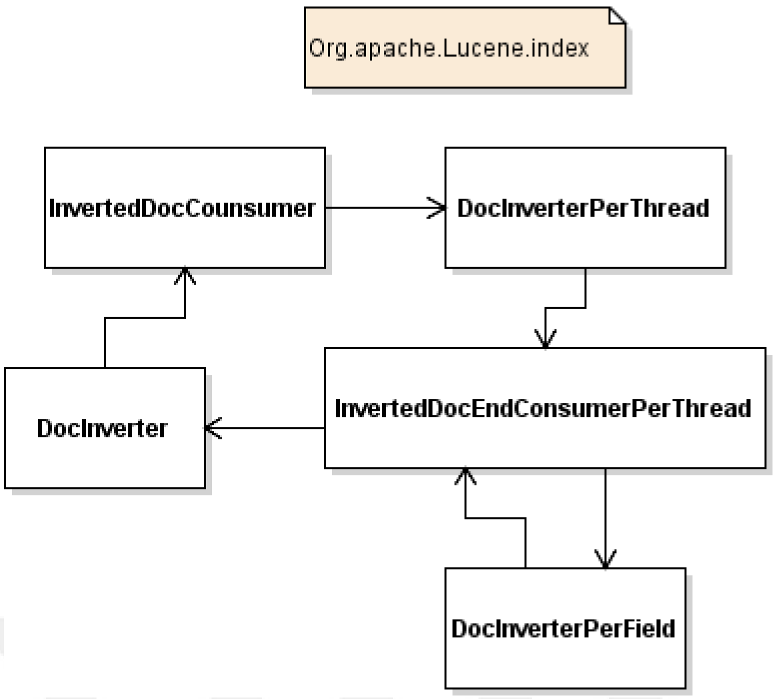

Figure B.2 Lucene 3.3.0 Cycle 2

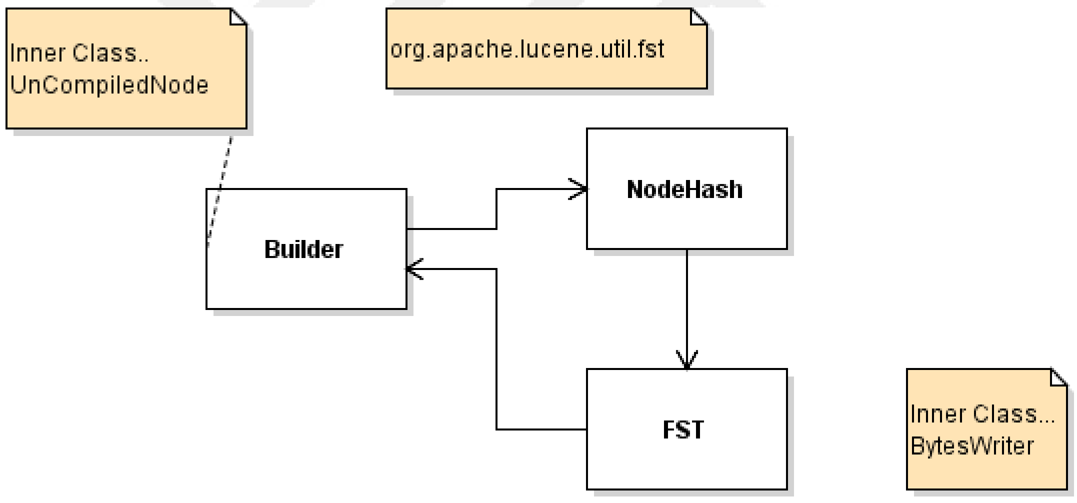

Figure B.3 Lucene 3.3.0 Cycle 3

APPENDIX-C

LUCENE 7.0.1

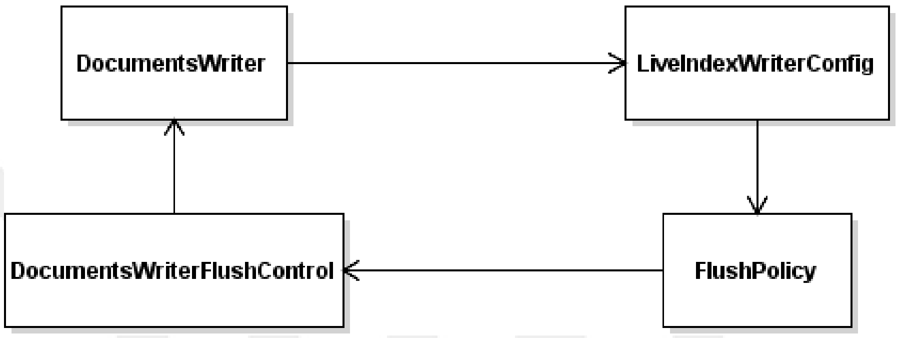

Figure C.1 Lucene 7.0.1 Cycle 1

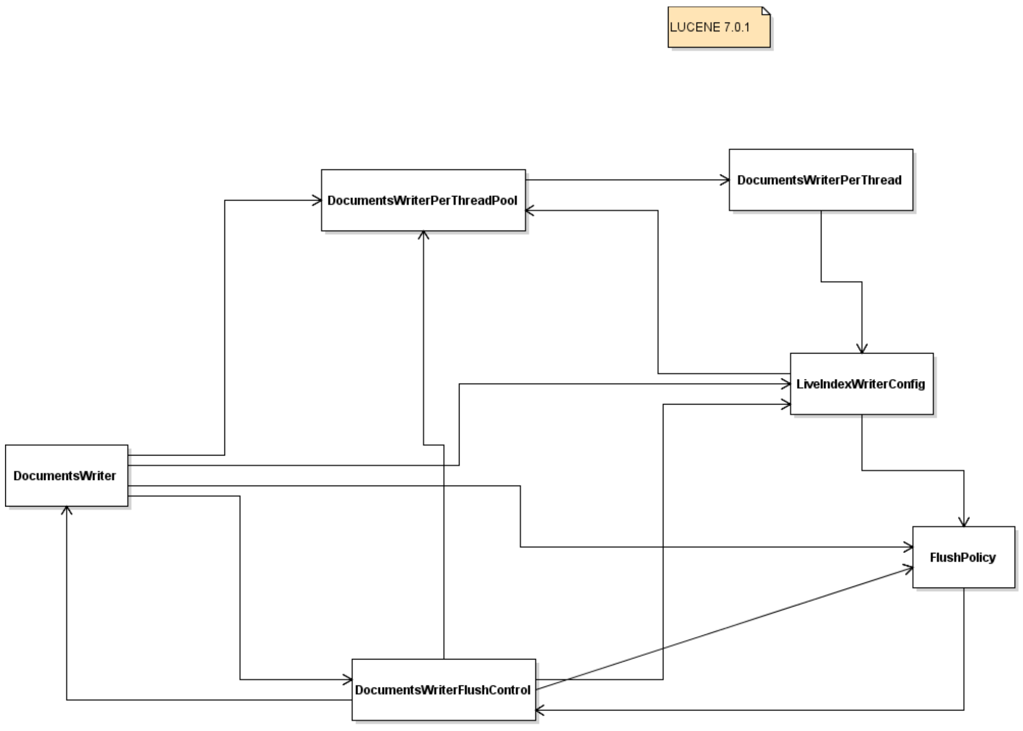

Figure C.2 Lucene 7.0.1 Cycle 2

CURRICULUM VITAE

PERSONAL INFORMATION

Name Surname	: Samer AL-Rubaye
Date of birth and place	:05/05/1983 Irak -Baghdad
Foreign Languages	:english
E-mail	:softieng@yahoo.com

EDUCATION

Degree	**Department**	**University**	**Date of Graduation**
Master			
Undergraduate	Computer engineering	University of technology	2004/2005
High School			

PUBLISHMENTS Conference Papers

1. 8th International Conference on Software Engineering and Service Science (ICSESS 2017)