An Empirical Study of the Correlation between Code Smells And Software Bugs

by

Gayathri Ganesan

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science (Software Engineering) in the University of Michigan-Dearborn 2018

Dedication

I dedicate this to God, my parents, my husband and all those who have helped me throughout the journey of my life.

Acknowledgements

I would like to take a moment and thank those who have helped me through the completion of my thesis.

Firstly, I express my gratefulness to my Professor Dr. Marouane Kessentini to have dedicated his time and energy guiding me through every step of the project. His motivation and witty ideas led to the start and end of the project.

I also thank all my lecturers who helped me throughout my master’s program and have provided valuable feedback on a timely fashion.

Finally, I would like to thank my family who has been a continuous support throughout this journey.

Thank you.

Table of Contents

List of Tables

List of Figures

Abstract

Bug predictions helps software quality assurance team to determine the effort required to test the software application. Anti-patterns and code smells can greatly influence the quality of the code. Refactoring can be a solution to remove the negative impact of these anti-patterns. In this paper, we explored the influence of code smells on the code and severity of bugs reported on multiple versions of the projects such as BIRT, Aspect J and SWT. We evaluated the correlation between the different code smells and severity of the bugs reported on these classes. This can help the quality assurance specialists and project managers assess the testing effort required based on the code smells detected. This can prove beneficial to the developers to restructure or refactor before deploying the code in the test environment. On the other hand, the testing team can concentrate on the bug prediction models, testing plan and assess the number of resources needed to perform testing. The empirical validation of our work found a strong correlation between several types of code smells and software bugs based on three large open source projects.

Chapter 1 Introduction

Many industries such as automotive, banks, insurance and health care are dependent on software systems. Ideally, software applications should not contain any bugs and if it does, it may prove to be very costly depending on its severity. Hence, it is absolutely important that these software applications function efficiently. Errors in the code need to be fixed based on its severity and priority. However, fixing these bugs prove to be very costly. It is estimated that 80% of the total cost of a software system is spent on fixing bugs [2]. To reduce this cost, many bug prediction models have been proposed by the research community to identify areas in software systems where bugs are likely to occur. The vast majority of these bug prediction models are built using the product (*e.g.*, code complexity) and process (*e.g.*, code churn) metrics, most of which are not actionable. For example, Nagappan and Ball [3] have used code churns to predict bugs in software systems [4].

Developers introduce code smells in the code due to lack of knowledge of design patterns, optimization techniques and efficient problem-solving capability. These antipatterns and code smells in the code might not affect the functionality of the system, but the code may be difficult to maintain. Software development guidelines are a set of rules which can help improve the quality of software. These rules are defined on the basis of experience gained by the software development community over time. Software antipatterns are a powerful and effective form of guidelines used for the identification of bad design choices and development practices that often lead to poor-quality software [7].

These code smells may also lead to bugs in the code. Previous work by Khomh et al. [5] have found that classes with code smells, are more prone to bugs than other classes[4]. In order to reduce the number of bugs, code smells must be reduced. This can be done through refactoring, and restructuring the code. There are various open source software and enterprise level software which can refactor and restructure the code to reduce the antipatterns and code smells.

If we can compute the relationship between code smells and defects, then a lot of stakeholders can benefit from the results. The developers can save time fixing bugs, the testers can save time from restesting the code, the business team and other stakeholders who have invested time and money will see results of the feature within the estimated time for completion of the project.

In this thesis, we have tried to compute the relationship between code smells and bugs reported by using out of the box machine learning algorithm provided by a tool named Weka. Machine learning and correlation computation were also done using the algorithms provided by the language R. Weka and R were utilized to observe a correlation between severity of the bugs reported and various code smells in the code. This study was conducted using the data extracted from three open projects such as BIRT, SWT, Aspect J. And, we have tried to answer the below research questions.

RQ1. Does code smell affect the severity of the bugs?

We found that some of the code smells have more impact on the severity of the bugs than others.

RQ2. Can these machine learning algorithm results provide us more information?

Not only the severity of the bugs is affected, the overall number of bugs reported are impacted by the code smells present in the code.

The rest of the sections in the paper are organized as follows. Chapter 2 states some of the literature work already performed on learning from code smells and antipatterns in the code. Chapter 3 covers the methodology followed by us to find the correlation between code smells and severity of the bugs. Chapter 4 describes the results of our study. We have also explained the threats to validity. Chapter 5 states the conclusion and plans about future work.

Chapter 2 Literature study

This section covers the literature study on finding the relationship between code smells and bugs reported and other related work.

Zhang et. al. in their 2017 publication [8] have researched mainly about three 3 types of bad design features on 18 versions of the Apache common series. They discuss about the number of defects in the source files could have which has the 3 kinds of bad designs. They found that these have an impact on the number of defects reported.

The research based on predicting the high and low severity faults were conducted by Zhou et. al. [9]. They have considered the object oriented metrics such as CBO, WMC, RFC, LCOM, DIT and used machine learning methods such as Naïve bayes, random forest and NNge to find the correlation with the low and high severity of the bugs. They found that the CBO, WMC, RFC, and LCOM metrics have significance across defect severity. DIT metric did not have any significant impact on the severity of the defects. Subramanyam and Krishnan analyzed an e-commerce application developed in C++ and Java [22]. The experiment was based on the application to study how the size of the class affects faults. The study was performed on 405 C++ and 301 Java classes and how the metrics were related to the faults irrespective of the size of the application.

Shatnawi et al [10] in their paper talked about the software metrics and error proneness during the implementation phase of the development lifecycle. They considered the antipatterns, cyclic dependencies and coding methodology to determine the defect proneness.

Khomh et al. investigated the relation between antipatterns and defects reported. They performed the analysis on 10 releases of ArgoUML, 13 of Eclipse, 18 of Mylyn, and 13 of Rhino, and considered the changes and defects occurring between the releases. In their paper, they considered 13 antipatterns which are AntiSingleton, Blob, ClassDataShouldBePrivate, ComplexClass, LargeClass, LazyClass, LongMethod, LongParameterList, MessageChain, RefusedParentBequest, SpaghettiCode, SpeculativeGenerality, SwissArmyKnife. They found Complex class, Lazy class, Message chain, Long method and Anti- Singleton had more than 55% correlation with fault proneness.

Sabane et al. [12] considered four projects to study about their antipatterns and how it affects unit testing and test-cases. The authors in this paper have performed investigation and found that the classes needed more unit testing when antipatterns found are more. A high number of test cases were required for the complex classes. This is in comparison with CDSBP(ClassDataShouldBePrivate), LzC (LazyClass), LM (LongMethod), MC (MessageChains), RPB (RefusedParent- Bequest). In conclusion, the study finds out that the antipattern increases the number of unit testcases written.

Several studies results say that the code smells and anti patterns impact the quality of the software negatively [25, 26, 27, 28, 29, 30, 31, 32]. Deligiannis et al. [23] conducted a study to analyze how God classes impacted the software's maintainability. Their approach confirmed that higher the design quality, higher the maintainability and understandability. They also concluded that God classes had a negative impact on the quality of the code.

Abbes et al. [24] performed a few experiments on understandability of the projects with and without Blob classes, Spaghetti code and both Blob class and Spaghetti code in it, by student

They found the students found it more difficult to understand the code with significant amount of Blob classes when compared to those projects without them. However, irrespective of the presence of Spaghetti code, there was no significant impact on the understanding. However, there was an impact and a difference in understanding of the code when both Blob class and Spaghetti code were present. The combination of both made a significant difference in understanding it.

Olbrich et.al. [13] studied the impact of code smells on the quality of the code. They mainly focused on God class and Brain class. They conducted analysis on two open source and large scale projects. They found that these 2 code smells had a negative impact on the number of bugs reported. But, when they were normalized and without any of the mentioned code smells, the number of defects reported were less. The study also concluded that the classes with God class and Brain class do not affect the entire quality of the software unless they are large in size.

Chapter 3 Methodology

This section describes the methodology followed and this section aims to answer the research questions mentioned in section 1.

a) Data acquisition

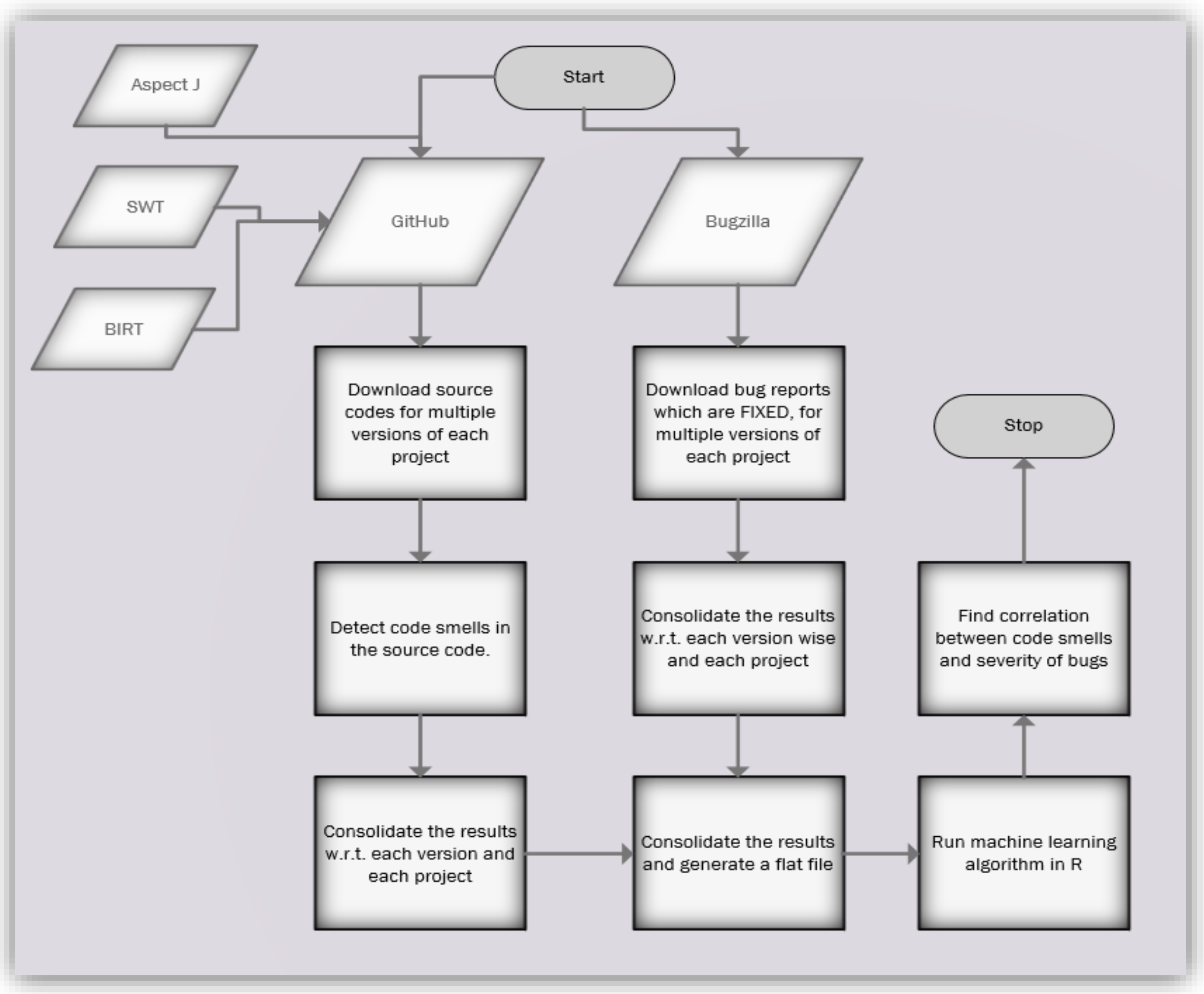
In this section, we go over the process of how and where the data was acquired for analysis. We used at-least 15 versions of the open source projects. The projects which were analyzed are SWT, BIRT, Aspect J. The source code for about 15 versions were downloaded from checked in versions in GitHub. The source code for all the projects mentioned in the below table are written in Java. Each project version was sent through code smells and antipattern detection tools. Below is the list of projects used for analysis and versions of code which were downloaded.

Project	**Versions**
SWT	15 versions between 0.9.0 and 4.2
BIRT	15 versions between 2.1.0 and 4.2.2
Aspect J	20 versions between 1.5.4 and 1.9.0

Table 1:List of projects considered for data analytics

Along with the source code, we downloaded the bug reports for each version mentioned above, for each project, from Bugzilla. These bugs were reported by various individuals, in the bug reporting tool. We downloaded only those bugs which were Fixed and closed. We did not consider any open bugs for analysis. These defects were either fixed in the next version or future versions of the code. Data processing

In this section, we discuss about the steps taken to transform the data captured in data acquisition section into more of a readable format by the tools used i.e. Weka and R. The below figure is the high level flow diagram of the study.



Aspect J
SWT
BIRT
Start
GitHub
Bugzilla
Download source codes for multiple versions of each project
Download bug reports which are FIXED, for multiple versions of each project
Stop
Detect code smells in the source code.
Consolidate the results w.r.t. each version wise and each project
Find correlation between code smells and severity of bugs
Consolidate the results w.r.t. each version and each project
Consolidate the results and generate a flat file
Run machine learning algorithm in R

Figure 1:Complete flowchart of the methodology used.

Step 1) *Downloading the source code-* We downloaded about 15 versions of the code, for both SWT and BIRT projects, from GitHub. For the Aspect J project, we downloaded about 20 versions of the project from GitHub. We considered only the Java files and the script files for analysis. So, the total number of versions of application source code considered for analysis added up-to 50.

Step 2) *Detecting antipatterns*- The downloaded application code for each version for each project were run through antipattern and code smell detectors, to detect the various code smells. The code smells detected by the tools are CyclicDependencies, BlobClass, GodClass, DataClass, SchizophrenicClass, RefusedParentBequest, DistortedHierarchy, IntensiveCoupling, ShotgunSurgery, FeatureEnvy, BlobOperation. This data is then used for data analysis. Once the antipatterns are detected for each class for each version of a given project, we find the sum of the antipatterns detected for each given version considered here for analysis, for each project. Below is a complete list of code smells mentioned above and its definitions.

ANTI PATTERN	**DEFINITION**
Cyclic Dependencies	A cyclic dependency is a relation between two or more modules which either directly or indirectly depend on each other to function properly. [14]
BlobClass	A class that contains almost all the functionality and a lot of responsibilities in a given application.
GodClass	A God Class is an object that controls way too many other objects in the system and has grown beyond all logic to become The Class That Does Everything.[15]
DataClass	A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters) [16]

SchizophrenicClass	A class that contains disjoint sets of public methods that are used by disjoint sets of client classes [17]
RefusedParentBequest	If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. [18]
DistortedHierarchy	A Distorted Hierarchy is an inheritance hierarchy that is unusually narrow and deep. This design flaw is inspired by one of Arthur Riel's heuristics, which says that "in practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six". Having an inheritance hierarchy that is too deep may cause maintainers "to get lost" in the hierarchy making the system in general harder to maintain. [19]
IntensiveCoupling	Intensive Coupling is the flaw of an method when a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes [20]
ShotgunSurgery	This smell is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior. Whenever a method is called by too many other methods, any change to such a method ripples through the design. Such changes are likely to fail when the number of to-be-changed locations exceeds the capacity of human's short term memory. [20]
FeatureEnvy	The Feature Envy design flaw refers to functions or methods that seem more interested in the data

	of other Classes and modules than the data of those in which they reside. These "envious operations" access either directly or via accessor methods. This situation is a strong indication that the affected method was probably misplaced and that it should be moved to the capsule that defines the "envied data" [20]
BlobOperation	A Blob Operation is a very large and complex operation, which tends to centralize too much of the functionality of a class or module. Such an operation usually starts normal and grows over time until it gets out of control, becoming hard to read and maintain [20]

Table 2:List of code smells and its definitions

Step 3) *Extracting bug reports*- For each version of the project mentioned in the previous section, resolved bugs with severity of the bugs reported are collected from Bugzilla. These bugs are resolved and fixed. Moreover, the severity level of the bugs collected for analytics are showstopper, critical, major, normal, minor. Showstopper being the most critical.

Apart from the above data collection and pre-processing, the total number of bugs reported were also considered as part of the data acquisition process. Figure 2 depicts the high-level view of the number of bugs reported for each project for some of the project releases.

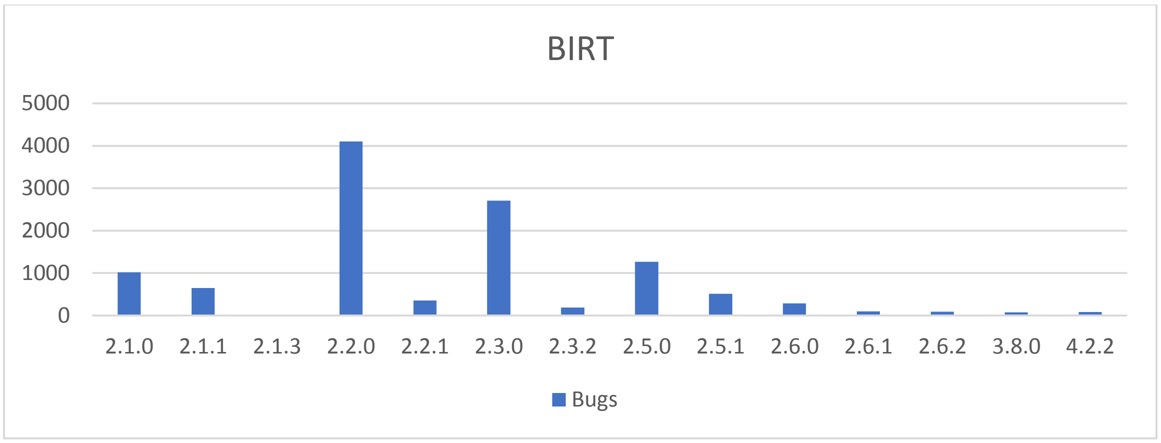


Figure 2: Charts depicting bugs reported in the major versions of the BIRT project

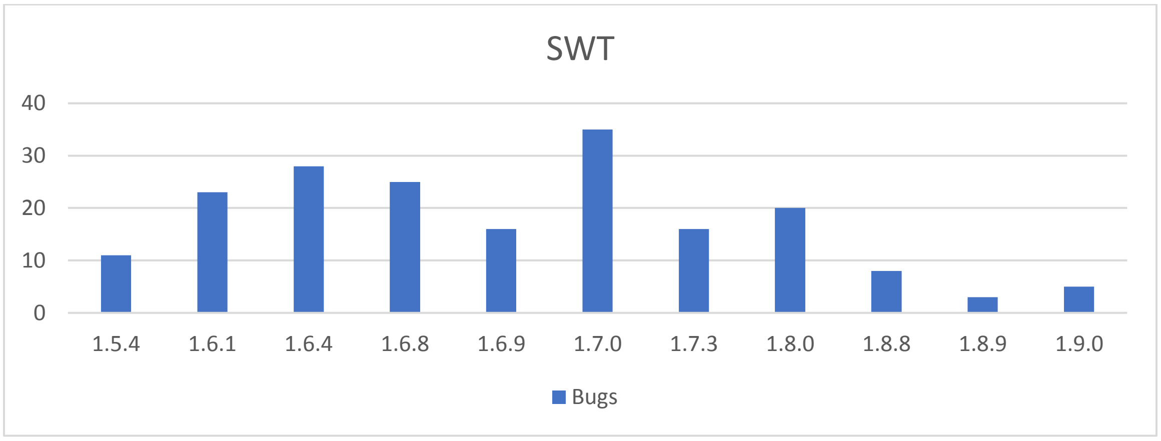


Figure 3:Charts depicting bugsreported in the major versions of the SWT project

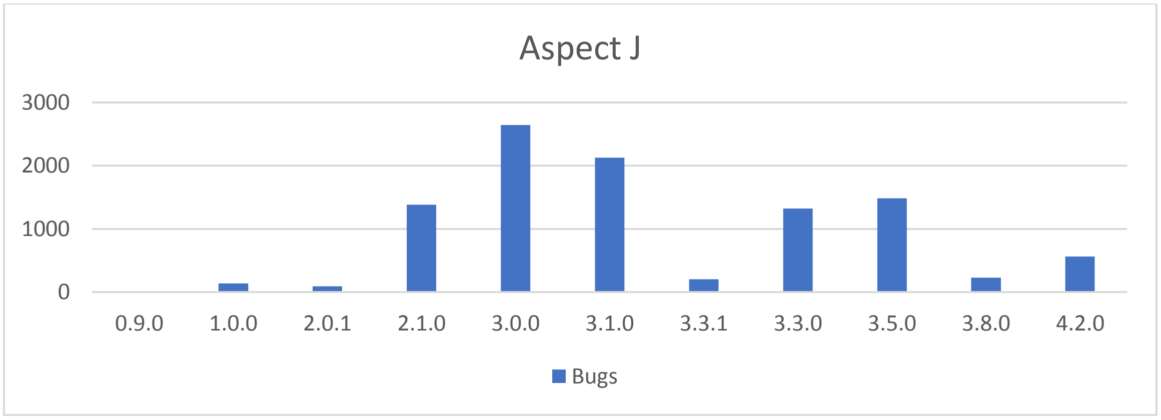


Figure 4:Charts depicting bugs reported in the major versions of the Aspect J project

b) Machine learning tools used

In this section, we discuss about the algorithms and methods used for machine learning analysis and finding correlation. R’s Linear regression algorithm was used to perform data analytics and run machine learning algorithms. In order to find the correlation between the severity of the bugs and antipatterns detected in the source code, there were 2 algorithms provided by R out of the box was utilized. The algorithms used are Kendall and Spearman and average was computed. We also ran one of the machine learning algorithms to find correlation provided by Weka[6], purely for our reference.

c) Analysis methodology

A correlation coefficient measures the extent to which two variables tend to change together. The coefficient describes both the strength and the direction of the relationship[21]. The results provided by the outcome of the machines learning algorithms depicts the correlation between the number of severity of bugs reported and detected code smells. The correlation was then converted into percentages. To verify our analysis and performance of correlation algorithms, we compared the results with the latest version results. For instance, if the correlation between critical bug and shotgun surgery antipattern is around 50%, then we took the test data from the latest version of the 3 open source projects and compared with the analysis results. Below are the correlation results of three open source projects BIRT, SWT and Aspect J with the severity of the bugs. The average of the results was found, and they were considered as the final correlation results between code smells and severity of bugs.

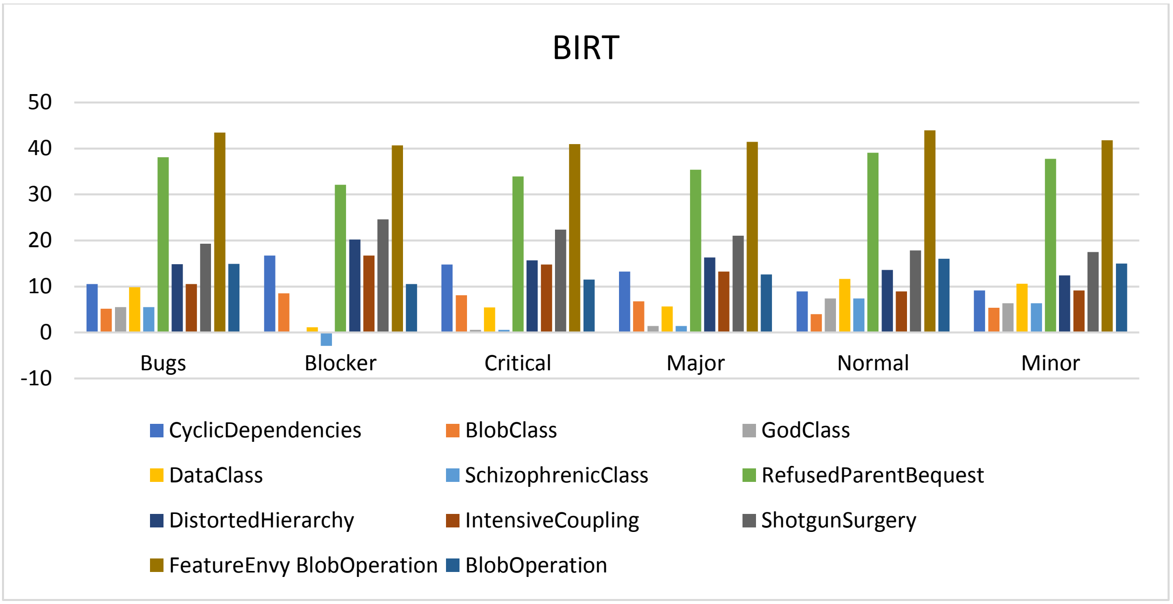


Figure 5:X Axis indicates the severity of the bugs. Y axis indicated the correlation of the bugs. Color indicates the type of code smell for project BIRT

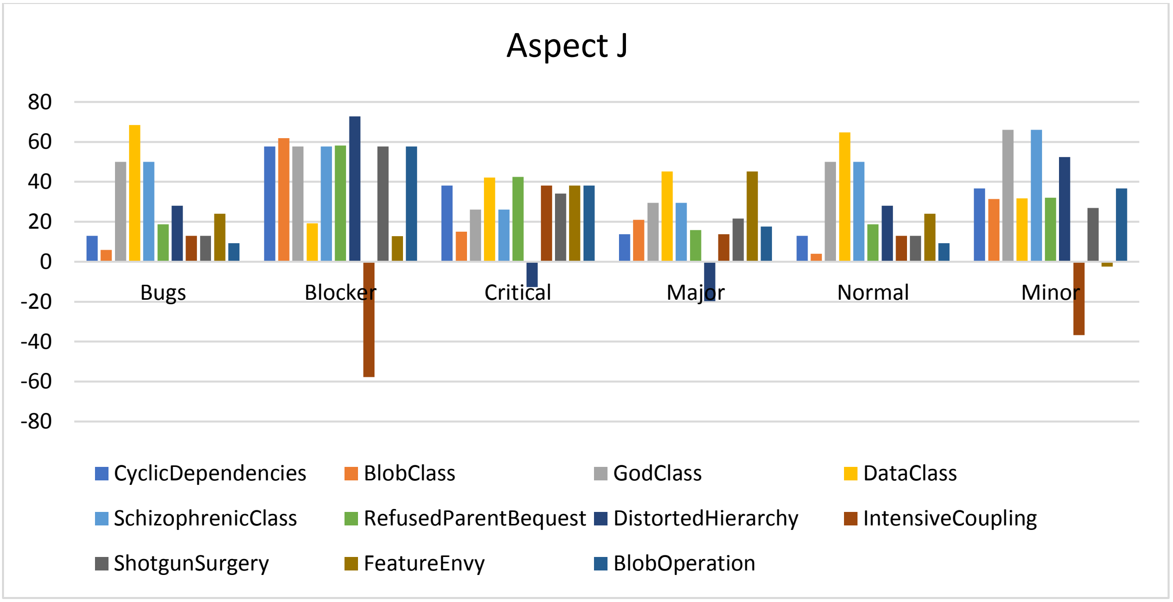


Figure 6:X Axis indicates the severity of the bugs. Y axis indicated the correlation of the bugs. Color indicates the type of code smell for project Aspect J

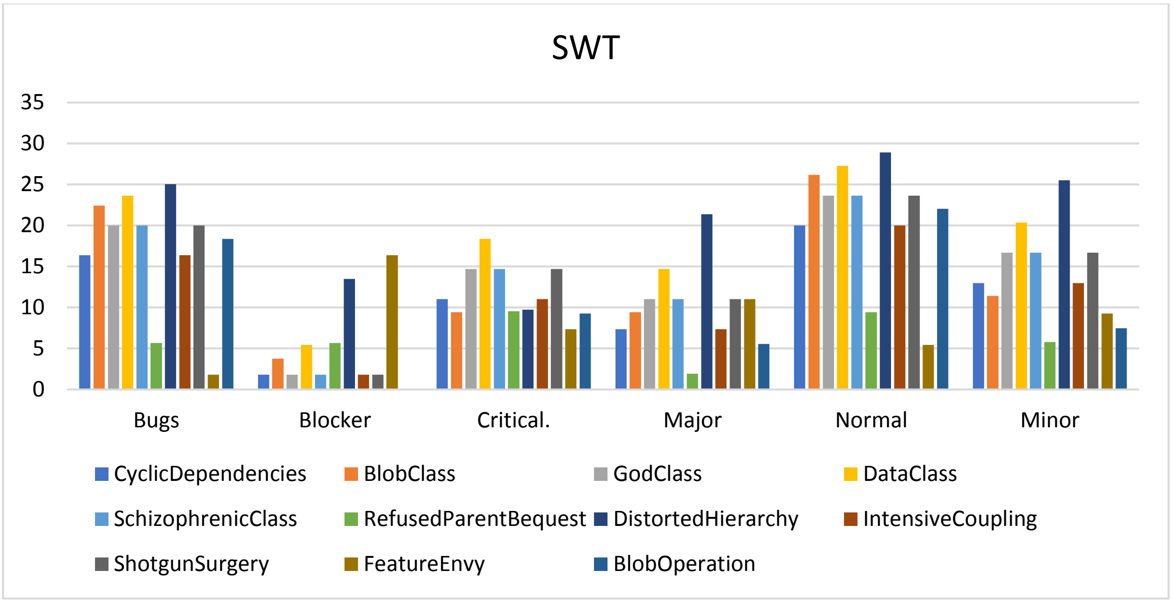


Figure 7:X Axis indicates the severity of the bugs. Y axis indicated the correlation of the bugs.

Color indicates the type of code smell for project SWT

Chapter 4 Results

RQ1. Do antipatterns affect the severity of the bugs?

Some antipatterns present in the application have adverse effect on the severity of the bugs reported. Firstly, we shall go through the results for each project individually. For project *BIRT*- Code smells such as Feature envy and RequestParentBequest have a major impact on the severity of the bugs overall when compared to other code smells. Bugs with severity level such as blocker, critical, major, minor, normal are equally affected and have a high correlation with Feature Envy code smell. Bugs with normal severity are reported more when RequestParentBequest are high in the code. In general, when both feature envy and RequestParentBequest are higher, the normal severity of bugs are higher in most of the cases considered during the study.

For project Aspect J, Shotgun surgery and Blob class has a major impact and have higher correlation on the Blocker bugs in most of the versions. Intensive coupling had a negative correlation with Blocker, minor severity bugs and so is Distorted Hierarchy code smell on major severity of bugs.

For project SWT, Distorted Hierarchy had the highest correlation with normal severity bugs. Data class has the second highest correlation with minor and normal severity bugs. Shotgun surgery has higher correlation with normal, critical and minor severity bugs.

Now, let us look at the antipatterns which has major impact on the bugs when all the data was inputted through the machine learning tools. Antipatterns such as Shotgun surgery have a huge impact mainly on the critical bugs reported. They have a correlation of nearly 47% with the critical bugs. That means, more the antipattern found, more will be the critical of bugs detected. Overall, shotgun surgery antipattern is correlated to the number of bugs, blocker, critical, major, normal and minor bugs.

Next comes the Data class antipattern. This antipattern has a potential to introduce bugs which could be a blocker or show stopper. There is nearly a 38% correlation. Between data class and blocker bugs. Same is the case with Blob class. Blob class when und in a version of the code, then there is a good possibility that there that class may contain critical or blocker bugs. Other mediocre impacts on the bugs were from antipatterns such as Feature Envy, Data class on number of bugs, major, normal and minor bugs reported.

RQ2. Can these antipatterns results provide us more information?

Not only the severity of the bugs is affected, the overall number of bugs reported also depend on the antipatterns found in the code. Looking at the results, some of the antipatterns have an impact on the number of bugs reported. Shotgun surgery, Feature envy, Blob operation, Blob class and data class are some of them.

The correlation results obtained can be utilized widely only when there is cross functional
use for it. Projects which are developed for other domains and in other platforms should also be able to use these results. Training data might not be available for all the projects and hence an analysis of this sort is difficult to perform. In Seyyed Ehsan et.al. [4], the authors have investigated to what extent one can use cross-system antipattern information to predict bugs. The table bellows show us the average results of correlation obtained from algorithms in R and Weka.

Correlation in percentage

Type of Code Smell	**Bugs**	**Blocker**	**Critical**	**Major**	**Normal**	**Minor**
CyclicDependencies	15.262321	16.97536	20.96995	16.22578	16.41439	5.789681
BlobClass	33.690053	**35.43676**	**36.6478**	34.04369	34.0954	26.627642
GodClass	38.750997	36.15608	34.69065	11.5761	9.88837	2.6435106
DataClass	25.755167	**37.94491**	32.26146	29.07789	27.25107	25.970856
SchizophrenicClass	8.750997	16.15608	14.65065	19.8461	9.83463	2.648818
RefusedParentBequest	11.332892	13.03306	16.03328	12.41961	12.48002	1.494728
DistortedHierarchy	24.314972	28.69896	27.81644	25.22981	25.51821	17.14929
IntensiveCoupling	15.262321	16.97536	20.96995	16.22578	16.41439	5.789681
ShotgunSurgery	**43.561208**	**40.60772**	**46.77912**	**42.8939**	**42.86865**	**42.181963**
FeatureEnvy	29.570747	34.94926	33.55192	31.64831	30.11961	23.985822
BlobOperation	25.616555	28.48134	31.47996	27.0109	26.79429	16.224009

Table 3:Correlation between code smells and severity of bugs measured in percentages

In order to determine the accuracy of the results, the number of bugs reported in the last 2 versions of the considered four open source projects were taken. When the code smell Shotgun surgery is high in a project, the number of critical bugs reported are more and vice versa. The major impact is seen on the critical bugs reported followed by the number of bugs reported in total.

Our future work is to perform the same experiment with other bigger open source projects from different domains and validate our results. This will provide us with more evidence that the results can be generalized for other projects as well irrespective of the domain.

Chapter 5 Threats to validity

A lot of practitioners and software researchers have studied about relation between antipatterns and bugs reported, code smells and number of bugs. This paper deals with code smells and severity of bugs in particular. In our study, we have considered four major software applications related to varied domain. We haven't considered any other application which is mainly utilized by healthcare, insurance, banking and other major industries. Hence, we cannot be sure that the results observed during this study applies across all the other domains.

In this section, we find out the factors which can bias our study, discuss about the construct validity threats, threats to internal validity, threats to external validity and reliability validity threats.

Construct validity threats concern the relation between the results which are observed and theory. In this study, we have considered only those bugs which were reported via Bugzilla and status of the bugs as Fixed or Closed. We did not consider open bugs or bugs marked as Others or enhancements.

Threat to internal validity is about the project we chose for our study, machine learning analysis methods used, deductions and conclusions obtained from them. The projects we chose were the open source projects from GitHub. The machine learning algorithms used to study the training data and find the correlation, were open source projects.Since, the results were compared with the results provided by language R and by Weka, any other detection techniques should be able to confirm our results.

Threat to external validity is to generalize our results to other domains. We have considered multiple versions of 3 open source systems with different sizes and belonging to different domains. We have considered only few code smell types and not a comprehensive list of all code smells types. Other researchers can analyze the code using various other code smell types. Also, further validation can be done using other set of machine learning software and algorithms which are more efficient in learning the data.

Conclusion validity threats is related to the methodology used and the outcome of the analysis. We haven't deviated from our initial research questions, assumptions and methodology used. Also, our analysis does not require to make assumptions about the data. Finally, we tried to provide all the details related to this study so that other researchers can replicate the study.

Chapter 6 Conclusion

Our study investigated the effectiveness of code smells on severity of the bugs. We detected the code smells from various open source java projects namely BIRT, SWT and Aspect J. Then, we downloaded the bugs that were reported and were later fixed. Finally, we ran machine learning methods to find the correlation between the two. Our findings strongly support the following:

1. We provided an empirical validation that some code smells can help us foretell severity of bugs which might be reported.
2. Some code smells, specifically Shotgun surgery and Blob class have a higher correlation with the bug which are a show-stopper and critical to the application. Data class and Blob class has a higher impact on the certain severity levels of bugs namely, critical and major. Whereas, Schizophrenic Class code smell has overall less impact.

References

[1] Rubin, Elyse. " The Dissertation Handbook: A Guide to Submitting Your Doctoral Dissertation and Completing Your Doctoral Degree Requirements." Diss. U of Michigan, 2017

[2] N. I. of Standards & Technology, “The economic impacts of inadequate infrastructure for software testing,” May 2002, uS Dept of Commerce.

[3] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in Proceedings of the 27th International Conference on Software Engineering, ser. ICSE ’05. New York, NY, USA: ACM, 2005, pp. 284–292.

[4] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan,”Predicting Bugs Using Antipatterns”

[5] F. Khomh, M. D. Penta, Y.-G. Gue ́he ́neuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” Empirical Softw. Engg., vol. 17, no. 3, pp. 243–275, Jun. 2012.

[6] https://www.cs.waikato.ac.nz/ml/weka/

[7] T. K. Das and J. Dingel, "State machine antipatterns for UML-RT," *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Ottawa, ON, 2015, pp. 54-63.

[8] X. Zhang, Y. Zhou and C. Zhu, "An Empirical Study of the Impact of Bad Designs on Defect Proneness," 2017 International Conference on Software Analysis, Testing and Evolution (SATE), Harbin, 2017, pp. 1-9

[9]- Y. Zhou, H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults", IEEE Transactions on software engineering, no. 10, pp. 771-789, 2006.

[10] - R. Shatnawi, W. Li, The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. Journal of systems and software, no. 11, pp. 1868-1882, 2008.

[11] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness", Empirical Software Engineering, no. 3, pp. 243-275, 2012.

[12] - A. Sabané, M. Di Penta, G. Antoniol and Y. Guéhéneuc, "A Study on the Relation between Antipatterns and the Cost of Class Unit Testing," 2013 17th European Conference on Software Maintenance and Reengineering, Genova, 2013, pp. 167-176

[13] S. M. Olbrich, D. S. Cruzes and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," 2010 IEEE International Conference on Software Maintenance, Timisoara, 2010, pp. 1-10.

[14] https://en.wikipedia.org/wiki/Circular_dependency

[15] http://wiki.c2.com/?GodClass

[16] https://refactoring.guru/smells/data-class

[17] https://www.coursehero.com/file/p2rrii9/The-Schizophrenic-Class-anti-pattern-79-Problem-description-A-class-that/

[18] https://refactoring.guru/smells/refused-bequest

[19] Riel, A. J. (1996). Object-oriented design heuristics (Vol. 338). Reading: Addison-Wesley.

[20] Umme A, Iftekhar A, Rana A, Danny D, Carlos J(2016), "Understanding Code Smells in Android Applications"

[21] https://support.minitab.com/en-us/minitab-express/1/help-and-how-to/modeling-statistics/regression/supporting-topics/basics/a-comparison-of-the-pearson-and-spearman-correlation-methods/

[22] R. Subramanyan and M.S. Krisnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," IEEE Trans. Software Eng., vol. 29, no. 4, pp 297- 310, Apr. 2003.

[23] . S. Deligiannis, I. Stamelos, L. Angelis, , M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," Journal of Systems and Software, vol. 72, no. 2, pp. 129 – 143, 2004.

[24] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in Proceedings of the 2011 15th European Conference on Software Mainte- nance and Reengineering, ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 181–190.

[25] Wang, Hanzhang, Marouane Kessentini, and Ali Ouni. "Bi-level identification of web service defects." International Conference on Service-Oriented Computing. Springer, Cham, 2016.

[26] Mansoor U, Kessentini M, Wimmer M, Deb K. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. Software Quality Journal. 2017 Jun 1;25(2):473-501.

[27] Mkaouer MW, Kessentini M, Bechikh S, Cinnéide MÓ, Deb K. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empirical Software Engineering. 2016 Dec 1;21(6):2503-45.

[28] Ouni, Ali, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. "Web service antipatterns detection using genetic programming." In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1351-1358. ACM, 2015.

[29] Kessentini, M., Wimmer, M., Sahraoui, H. and Boukadoum, M., 2010, June. Generating transformation rules from examples for behavioral models. In Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications (p. 2). ACM.

[30] Kessentini, Marouane, Houari Sahraoui, and Mounir Boukadoum. "Example-based model-transformation testing." Automated Software Engineering 18, no. 2 (2011): 199-224.

[30] Ouni, Ali, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. "Multi-criteria code refactoring using search-based software engineering: An industrial case study." ACM Transactions on Software Engineering and Methodology (TOSEM) 25, no. 3 (2016): 23.

[31] Ouni A, Kessentini M, Sahraoui H, Inoue K, Hamdi MS. Improving multi-objective code-smells correction using development history. Journal of Systems and Software. 2015 Jul 1;105:18-39.

[32] Kessentini M, Bouchoucha A, Sahraoui H, Boukadoum M. Example-based sequence diagrams to colored petri nets transformation using heuristic Search. InEuropean Conference on Modelling Foundations and Applications 2010 Jun 15 (pp. 156-172). Springer, Berlin, Heidelberg.