EXPLOITING ABSTRACT SYNTAX TREES TO LOCATE SOFTWARE DEFECTS

by

THOMAS JOSHUA SHIPPEY

Submitted to the University of Hertfordshire in partial fulfilment
of the requirements of the degree of

DOCTOR OF PHILOSOPHY

School of Computer Sciences
University of Hertfordshire

May 7, 2015

Acknowledgements

Who would have thought that an interval in a Vancouver Canucks NHL game would have led me to undertaking a PhD? Not me. 2am boredom led me to have a look at some code that created a digital clock. The Java clock had been developed by David Bowes earlier in the day. It had a alarm that let out a dull beep when a certain time had been reached. I thought, surely this can be improved? So it took me the next two periods of the Canucks game to modify the alarm clock so it played songs instead of a beep when the alarm was set off. Upon showing this to David, he seemed impressed. Ultimately, this led to David supervising my Masters project and the rest they say, is history. So, my first acknowledgement goes to my principal supervisor David Bowes. Without your encouragement and belief in me, I would not have started on this journey. You have been fantastic over the past three years (four if you include the Masters!). Your passion for the subject is infectious and your enthusiasm has compelled me to always want to impress you with new work each week. It has been a real pleasure to work with you over the course of this PhD.

My second acknowledgement goes to my other supervisors; Tracy Hall and Bruce Christianson. You both have been a huge support in the past three years. Tracy, thank you for helping me develop the rigour needed to undergo a PhD. I must thank you for being patient as my writing style went from bad, to ok, to acceptable, to good. I hope that I have finally figured it out and hopefully your red pen can have some well earned rest. Bruce, your experience and guidance has been invaluable and it has been a pleasure to work with you over the past few years. Thirdly, I would like to thank everyone that I have met in the STRI and Computer Science department for helping me through my PhD years.

I would like to thank my mum, Ruth Shippey for everything that she has done for me over the last 28 years. Quite literally, without you I would not be in the position to do so well in life. Your support and guidance over the years has been amazing. You have created a great home and loving atmosphere that has allowed me to achieve what I have achieved so far. I know the last nine years have been hard, but you have been the rock I have been able to lean on, even when I know you are crumbling. You are the best mum I could have asked for, I hope you are proud of what I have achieved. I would like to thank my sister, Amy Shippey for all her support in the last couple of years. Thank you for being the annoying little sister that made me hide away in my room and therefore let me get my work done (only kidding!). Thank you to my other sister, Catherine. You only came in to my life during the PhD but I couldn't ask for a better older sister. Your love and help has been invaluable. Thank you to all my friends and family, for all your help and understanding during the last three years. The laughs you provide have helped keep me going. Special mention has to go to the Bradford WhatsApp group for allowing me to vent when Arsenal and Arsene decide to ruin the weekend. I am going to have to think of a better excuse to bail out of events now that the PhD is over!

Thank you Katie Steingold. Thank you for all your love, support and encouragement. Thank you for being there for me when I need it most. I hope I can repay you in the years to come.

Finally, I want to thank my dad, Phil Shippey. Unfortunately, you will not be around to see what I have achieved. There is not a day that goes by that I do not think of you. Without the love and support you gave to me, I would not be a man capable of completing a PhD thesis. You always said, you did not know how to be a good father as you did not have one yourself. You never had to worry about that, you were the best I could have ever asked for. Wherever you are, I do hope you are proud of me.

ABSTRACT

Context. Software defect prediction aims to reduce the large costs involved with faults in a software system. A wide range of traditional software metrics have been evaluated as potential defect indicators. These traditional metrics are derived from the source code or from the software development process. Studies have shown that no metric clearly out performs another and identifying defect-prone code using traditional metrics has reached a performance ceiling. Less traditional metrics have been studied, with these metrics being derived from the natural language of the source code. These newer, less traditional and finer grained metrics have shown promise within defect prediction.

Aims. The aim of this dissertation is to study the relationship between short Java constructs and the faultiness of source code. To study this relationship this dissertation introduces the concept of a Java *sequence* and Java *code snippet*. Sequences are created by using the Java abstract syntax tree. The ordering of the nodes within the abstract syntax tree creates the sequences, while small subsequences of this sequence are the code snippets. The dissertation tries to find a relationship between the code snippets and faulty and non-faulty code. This dissertation also looks at the evolution of the code snippets as a system matures, to discover whether code snippets significantly associated with faulty code change over time.

Methods. To achieve the aims of the dissertation, two main techniques have been developed; finding defective code and extracting Java sequences and code snippets. Finding defective code has been split into two areas - finding the defect fix and defect insertion points. To find the defect fix points an implementation of the bug-linking algorithm has been developed, called S_e^+. Two algorithms were developed to extract the sequences and the code snippets. The code snippets are analysed using the binomial test to find which ones are significantly associated with faulty and non-faulty code. These techniques have been performed on five different Java datasets; ArgoUML, AspectJ and three releases of Eclipse.JDT.core

Results. There are significant associations between some code snippets and faulty code. Frequently occurring fault-prone code snippets include those associated with identifiers, method calls and variables. There are some code snippets significantly associated with faults that are always in faulty code. There are 201 code snippets that are snippets significantly associated with faults across all five of the systems. The technique is unable to find any significant associations between code snippets and non-faulty code. The relationship between code snippets and faults seems to change as the system evolves with more snippets becoming fault-prone as Eclipse.JDT.core evolved over the three releases analysed.

Conclusions. This dissertation has introduced the concept of code snippets into software engineering and defect prediction. The use of code snippets offers a promising approach to identifying potentially defective code. Unlike previous approaches, code snippets are based on a comprehensive analysis of low level code features and potentially allow the full set of code defects to be identified. Initial research into the relationship between code snippets and faults has shown that some code constructs or features are significantly related to software faults. The significant associations between code snippets and faults has provided additional empirical evidence to some already researched bad constructs within defect prediction. The code snippets have shown that some constructs significantly associated with faults are located in all five systems, and although this set is small finding any defect indicators that transfer successfully from one system to another is rare.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"They don't make bugs like Bunny anymore."

– Olav Mjelde

Defective software code is expensive for the software industry in terms of money, time, effort and loss of reputation. It has been estimated that around $312 billion per year is spent on finding and reducing the number of defects in software systems [Brady 2013]. Software faults can also effect a user's confidence in a system or product, for example; the opening of the US Affordable Health Care Act website, which damaged the reputation of the Act and the US president [Connor 2013], or when Samsung had to pull an Android 4.3 software update for its Samsung S3 mobile product. The software update caused a number of problems, for example an increase in battery drain, applications not working and alarms not activating [BBC 2013].

Software defect prediction is an important area of research trying to identify defective software code. Early identification of defects[1] may help reduce the costs caused by defects. This is because the costs associated with fixing defects increase as system maturity increases [Pressman 2001]. Defect prediction uses algorithms to create models which predict the areas of software code where defects are likely to occur. It is a well researched area, with over 200 studies published between 2000 and 2010 reporting software defect prediction models [Hall et al. 2012]. Despite software prediction being a well researched area, existing defect prediction models are thought to have reached a predictive performance ceiling [Menzies et al. 2010].

Software defect prediction models rely on independent and dependent variables. Independent variables are normally software metrics collected from software systems, for example software metrics include software code metrics (SCM) and process metrics. Examples of SCMs used in the models include Lines of Code (LOC) [Fenton and Ohlsson 2000, Mende and Koschke 2009, Weyuker et al. 2010, Zhou et al. 2010], object oriented metrics [Arisholm et al. 2010, Bird et al. 2009b, Cruz and Ochimizu 2009, D'Ambros et al. 2010, Khoshgoftaar et al. 2010] and complexity measures [Mende and Koschke 2010, Menzies et al. 2007, Turhan et al. 2009]. Process metrics include code churn [Hassan 2009, Khoshgoftaar et al. 1996, Nagappan et al. 2010, Nagappan and Ball 2005, Purushothaman and Perry 2005, Śliwerski et al. 2005], revision control histories [Bell et al. 2006, Graves et al. 2000, Hassan and Holt 2005, Moser et al. 2008, Ostrand et al. 2005, Weyuker et al. 2006] and number of previous faults identified [Hassan and Holt 2005, Kim et al. 2007]. The dependent variables in the models are normally fault variables (e.g. the number of faults predicted in a module, or if a fault has been predicted in the module).

Traditional SCM have been extensively used but most of the metrics introduced above suffer from being very coarse grained with capability to measure only a small sub-set of code features. Gray et al. [2011]

[1] The IEEE definition IEEE [2010] of a fault being a reported defect, where a defect is a mistake in code made by a developer which may result in a failure of the program to execute as planned is used throughout this dissertation.

suggest that the coarse grained nature of such metrics prevents machine learning techniques from effectively differentiating between defective and non-defective modules: if one module has the same metric values as another (say in terms of LOC), but they have not been labelled the same in terms of their defectiveness, this will hinder the learning algorithm's ability to learn. Gray et al. [2011] identifies many modules in the NASA datasets[2] which have identical values across a range of metrics but different defectiveness labels. This suggests that commonly used metrics are not sufficient to differentiate modules for defect prediction.

The natural language of software code has been used to identify defects. Natural language analysis is the study of the linguistic quality of the source code text. Poor language used by developers for creating code could harm the quality of the code and could hinder program comprehension [Abebe et al. 2012]. Program identifiers were investigated by Abebe et al. [2012] and Binkley et al. [2009] who both showed that the use of language processing measures were useful in defect prediction. Character analysis has also been used to associate text with defective modules [Zeller et al. 2011]. Detection using patterns of words in faulty modules was developed by Mizuno et al. [2007], where a fault-proneness filtering technique was developed based on spam filtering approaches. The use of text has problems due to differing languages spoken by different developers and the analysis of text has to have custom-made lexers to break it down into smaller pieces. This dissertation is interested in the the key building blocks of the source code, rather than the linguistic properties of the written code.

Defective code is not limited to software engineering. In biology, problems involving living organisms can be related to defective 'code'. Scientists investigate genetic disorders (defects in the living organism) using chromosomes. A chromosome is *"an organised package of DNA found in the nucleus of a cell"* [Institute 2014]. Humans have 23 different pairs of chromosomes and these chromosomes were used to discover if the existence/non-existence of a certain chromosome was associated with a particular genetic disorder (e.g. Down Syndrome is an example in which three copies of a chromosome exist, instead of two). This information could be used to predict whether a foetus or new born baby had a genetic disorder. Each chromosome is made up of DNA tightly coiled many times around proteins. It is this precise DNA sequence that allowed scientists to perform finer grained techniques to predict genetic disorders. DNA carries genetic information of living organisms [Insitute 2011]. The information is a sequence of four base pairs and this information is passed down from the parent organism to the child. The base pairs are the building blocks of the DNA and base pairs form between specific biological compounds called nucleobases. These base pairs are commonly known as adenine, guanine, thymine and cytosine and they are normally abbreviated to A, G, T and C. It is the order of the base pairs that determine the characteristics of a living organism (e.g. a sequence of TAAATGTCAA within a certain gene may indicate a person will have blue eyes). The greater the difference in the DNA sequence, the greater the difference in the organism. Small changes in the sequences that occur within the overall DNA sequence can be used to create a DNA profile. These DNA profiles can be used to identify correctly or incorrectly functioning genes within a living organism. Profiles are isolated and identified using chemical techniques, a process which produces a DNA fingerprint [Jeffreys et al. 1985]. Genes which do not work can be identified by comparing their profile (fingerprint) with that of a working gene. Although non functional genes may be due to small changes, the profiles can detect them. If we apply this analogy to software code, we can imagine that a small change to a piece of code could also impact on the correct functioning of that unit.

The aim of this dissertation is to develop an approach to identify features of software code, based on the principles of DNA profiling. This approach will create a new technique, by creating a profile based on

[2] http://nasa-softwaredefectdatasets.wikispaces.com/

the subsequences of particular pieces of code. The approach is based on analysing the Abstract Syntax Tree (AST) for Java code. The nodes in a Java AST contain Java kinds. These Java kinds[3] define the low level programming constructs that have been used in a piece of code and the order in which these are used. A method is turned into a sequence of kinds by walking its corresponding AST tree. The sequences are created by parsing the source code into an AST and then traversing the AST to gather the kind sequence. Figures 1.1, 1.2 and 1.3 and the Table 1.1 show how a piece of software code is transformed into a kind sequence. Figure 1.1 is a piece of code that describes a `Do While Loop` which prints of the value of x while the value is under 20. Figure 1.2 shows the AST which is created when this Do While Loop is parsed by the Java parser. Each white rectangle represents a different kind on the AST. This AST is then walked to create a sequence. The tree is walked using a left depth first algorithm (the number on the lines represent the order in which the tree is walked) each of the kinds is placed into a sequence. 1.3 shows the resulting sequence when the AST is fully walked. 1.1 shows which code is contained in each of the different kinds and thus how this code is broken down using the different kinds.

A code snippet is a subsequence of the main method sequence. These code snippets are extracted from the main method sequence by using a sliding window algorithm. Starting at the first kind in the method sequence and appending the kinds that follow up to a maximum code snippet length. For example, the first code snippet that would be extracted from the sequence in Figure 1.3 would be `DO_WHILE_LOOP`, the second `DO_WHILE_LOOP; BLOCK` and the third `DO_WHILE_LOOP; BLOCK; EXPRESSION_STATEMENT`. The length of the code snippet would increase by adding the next kind in the sequence until the maximum is reached. If, in this example that maximum was seven, the seventh code snippet would be `DO_WHILE_LOOP;BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; MEMBER_SELECT; IDENTIFIER;`. The algorithm would then return to the start and slide along one kind and start again. Meaning that the eight code snippet extracted would be `BLOCK` and the ninth `BLOCK; EXPRESSION_STATEMENT`. This would continue until all the possible code snippets are extracted from the sequence. A full example of how the code snippets are extracted from the main method sequence is found in Section 5.2. Code snippets capture the low level building blocks that have been used in the code and, so, provide comprehensive fine grained insight into the features of that code.

In this dissertation, the association between code snippets and faults is investigated, as well as the evolution of that association. Using three different software programs, this dissertation will show that there are code snippets that are significantly associated with faults in each of them. There are 201 code snippets that are significantly associated with faults across all of the datasets investigated. Although this is a small number, it is a significant result as code features that are associated with faults across systems is rare. The dissertation will also show that there are no code snippets significantly associated with non defective code in any of the systems analysed. When the evolution of code snippets is examined, it is found that the association between code snippets and faults does change as the systems evolve and faulty constructs are not being repeated over time.

[3] Appendix A.1 has the full list of Java kinds and their meanings.

```
1  do{
2        System.out.println("Value of x : " + x );
3        x++;
4     }
5  while( x < 20 );
```

Figure 1.1: A `DO_WHILE_LOOP` which prints out the value of x while x is less than 20. Figure 1.2 shows the AST walkthrough for this piece of code and Figure 1.3 is the sequence this code will make in AST kinds.

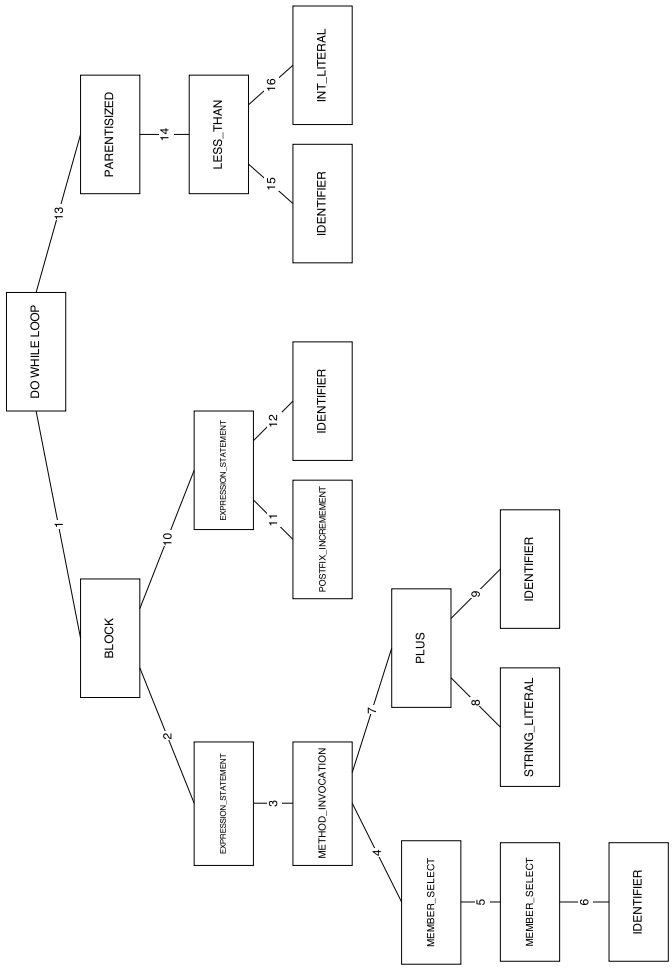


Figure 1.2: This figure shows the AST walkthrough for the `DO_WHILE_LOOP` shown in Figure 1.1. The number on the line represents the order in which the tree is traversed. Figure 1.3 shows the sequence that this traversed tree will make.

```
1  DO_WHILE_LOOP ; BLOCK ; EXPRESSION_STATEMENT ; METHOD_INVOCATION ;
2  MEMBER_SELECT ; MEMBER_SELECT ; IDENTIFIER ; PLUS ; STRING_LITERAL ;
3  IDENTIFIER ; EXPRESSION_STATEMENT ; POSTFIX_INCREMENT ; IDENTIFIER ;
4  PARENTHESIZED ; LESS_THAN ;  IDENTIFIER ; INT_LITERAL
```

Figure 1.3: The sequence of the `DO_WHILE_LOOP` in 1.1 and the traversed AST in 1.2 makes.

0	DO_WHILE_LOOP	`do { System.out.println("Value of x :  " + x); x++; } while (x < 20);`
1	BLOCK	`{ System.out.println("Value of x :  " + x); x++; }`
2	EXPRESSION_STATEMENT	`System.out.println("Value of x :  " + x)`
3	METHOD_INVOCATION	`System.out.println("Value of x :  " + x)`
4	MEMBER_SELECT	`System.out.println`
5	MEMBER_SELECT	`System.out`
6	IDENTIFIER	`System`
7	PLUS	`"Value of x :  " + x`
8	STRING_LITERAL	`"Value of x :  "`
9	IDENTIFIER	`x`
10	EXPRESSION_STATEMENT	`x++`
11	POSTFIX_INCREMENT	`x++`
12	IDENTIFIER	`x`
13	PARENTHESIZED	`(x < 20)`
14	LESS_THAN	`x < 20`
15	IDENTIFIER	`x`
16	INT_LITERAL	`20`

Table 1.1: A table showing how the AST is traversed. Each part of the tree is broken down into parts as it goes through the code. The numbers correlate to the order in which it traverses the tree shown in Figure 1.2.

1.1 Research Questions

This dissertation will answer two main research questions. One of the research questions is broken into two parts. The research questions are described in more detail below.

RQ1a: Are any code snippets significantly associated with faulty code? The discovery of small features of code that are associated with faults may be important. These features could be used to form a model to predict defects in other systems and future releases. These small features that are associated with faults could help identify bad features of code that are flagged as a potential problem for developers when they are developing a system.

RQ1b: Are any code snippets significantly associated with non-faulty code? If there are features of code that are associated with non-faulty code, this could indicate that the snippet is an example of good coding practice. The snippets associated with non-faulty code could also be used in a model to predict future defects in a system.

RQ2: Do the associations between code snippets and faultiness change as a system evolves? If a snippet significantly associated with faults appears in different releases of a system, this could indicate that it can be used to predict future defects within that system. If the snippet significantly associated with faults does not appear in all releases, this could indicate that significant snippets evolve as the system matures and that the code snippets are not a universal indicator of faults.

1.2 Contributions to Knowledge

This thesis makes the following contributions to knowledge:

1.2.1 Theoretical

This dissertation will present several theoretical contributions. The first contribution is the presentation of small code features to use within software engineering - *code snippets*. These snippets can be used in various different areas of software engineering (e.g. software evolution or code cloning). The thesis identifies snippets that are significantly associated with faulty code. This information could be used by programmers to find areas of code that could be potentially defective.

1.2.2 Methodological

This dissertation contributes a technique to find subsequences within software code which are significantly associated with defective/non-defective code. The technique also will find the most defective/non defective subsequences. This technique will allow future researchers to find their own subsequences in other programmes or to implement the technique in another programming language. This technique does not require manual intervention and can therefore be automated.

1.2.3 Practical

This dissertation provides a tool (ShipSZZ) which links defect data to code changes. The tool will also determine the insertion point of a particular defect. This tool has been created from scratch using Java. Chapter 4) has more information on bug linking and insertion finding. In addition to this, a full, manually checked, database of bug links for EJDT 3.0 is made available for other researchers to use. The database of linked faults extends the original work of Bird et al. [2009a], Śliwerski et al. [2005]. The results of this dissertation have been compiled into a journal paper which has been submitted to IEEE Transactions on Reliability [IEEEReliability 2015].

1.3 Structure of Dissertation

The following is an overview of the chapters that appear in this dissertation:

- **Chapter 2** gives the background on the main issues of the thesis. It will give an overview of what defects are, why they are a problem and how they are currently predicted. The chapter will describe DNA fingerprinting in more detail and how this relates to the work carried out in this dissertation.
- **Chapter 3** describes the methodology for the work undertaken in this dissertation. It will discuss the positives and negatives of the methodology chosen. The chapter will describe the datasets that have been used in the dissertation and introduce the various methods that are going to be implemented.
- **Chapter 4** describes the work completed in order to find defect fix and defect insertion points within a dataset. The chapter will show how the original SZZ algorithm with enhancements was used to search for defects within five different datasets. The chapter also presents how difficult it is to mine software repositories consistently and accurately.
- **Chapter 5** describes how Java code is sequenced and how these sequences are used to find Java snippets. The chapter presents the work undertaken to sequence Java code and how these sequences are used to find Java snippets that are associated with both defective and non defective code.
- **Chapter 6** describes the technique used to identify the statistical association between Java code snippets and faulty modules. The chapter provides statistical results to each of the research questions answered in this dissertation.
- **Chapter 7** describes the potential threats to the work that has been undertaken in this dissertation. The chapter will describe how these threats have mitigated during the research.
- **Chapter 8** discusses the results of the research questions that are answered in this dissertation. It will discuss the potential impact of the results that have been discovered across defect prediction.
- **Chapter 9** concludes this dissertation. The chapter will answer the research questions and highlight any contributions to knowledge that have been discovered.
- **Chapter 10** explores what future work could be undertaken on the basis of the work completed in this dissertation.

Chapter 2

Background

"Failure is a result Frasier, not a cause."

– Dr Lilith Sternin, 2001

This chapter will describe what a defect is, what defect prediction is and what methods are currently used to predict them. It will highlight the current limitations present in defect prediction and how this could potentially be overcome. Finally, this chapter will outline how DNA was first sequenced, how this motivated the work in the dissertation and how it will be related to software code.

2.1 Software Defect Prediction

A defect is defined as a shortcoming, imperfection or deficiency [CollinsEnglishDictionary]. IEEE [2010] define a defect within software as "an imperfection in a software product where the product does not meet its requirements or specifications". Defects are the result of errors which have been made by the development team during the creation of the software. For a defect to be known as a *fault* the error must have manifested itself during the use of the software product. The fault manifests itself when the software product does not perform in the way it was intended to when being used by the user. A defect is not known as a fault if it has been found during testing, or by a developer before the final implementation of a software product [IEEE 2010]. Figure 2.1 is a diagram (taken from IEEE [2010]) that presents the relationship between problems (errors), defects and faults. The diagram shows that a failure could be the result of problem with the system, a failure could caused by many problems, and many problems could produce the same failure. A fault is a specific type of defect that is discovered when a user experiences a failure when using a system. A fault could be the result of many failures and is a form of defect.

A software fault could be caused by a number of different factors. A software fault could be the result of a programming or design error, missing supporting libraries, or incompatibilities with an operating system. Some faults are trivial, for example, an incorrect hyperlink on a webpage or graphic glitches on a video game. However, some faults can cause major consequences. For example, Toyota (a car manufacturer) had various programming errors with its electronic throttle control system (ETCS) [Dunn 2013]. The programming errors were a key part of some of their cars automatically accelerating without the user pressing their foot on the pedal [Dunn 2013]. The automatic acceleration problem could have caused the deaths of up to 89 people [AssociatedPress 2010]. Software defect prediction uses models to predict the locations in a software system where errors are likely to have occurred. Defect prediction models use software metrics on which to base their decisions. Software metrics are a measure of some property of a particular piece of software, for example static code metrics or process metrics.

There are high costs involved with software failures [Brady 2013, Levinson 2001]. Pressman [2001]

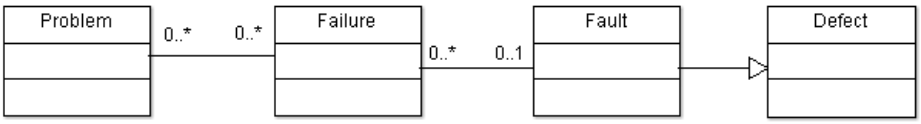


Figure 2.1: A diagram depicting the relationship between problems, failures, faults and defects. A fault is a subtype of a defect. A defect is a fault when the error manifests itself during software execution. Based on the diagram from IEEE [2010]

showed that the earlier a defect is fixed, the less cost involved in fixing said defect[1]. It is cheaper to repair a defect early, as there are extra costs involved during the different stages of development (e.g. if a defect is found after a product has been distributed to customers there are complaint resolution; product return; support and warranty costs involved) [Pressman 2001]. Figure 2.2 shows the relative cost of correcting an error in a software system [Pressman 2001] during the different stages of development. The bar chart shows that it can be up to 40-100 times more expensive to fix after the system has been released, compared to the requirements stage. Defect prediction is therefore important as it indicates potential areas that could contain defects, allowing time and resources to be assigned to these areas of a software system that have a greater propensity to defects [D'Ambros et al. 2010]. This could help save companies money as they are not fixing defects discovered later on in the development of a software system.

There are other costs involved with software defects which do not relate to financial costs. Defects can reduce the confidence a user has with a software system and/or what the system represents. For example, the user sign up problems during the initial phase of the ObamaCare website[2] registration, which opened in the USA in 2013. The ObamaCare website was meant to allow Americans in certain states to easily check if they are eligible for health subsidies after a new law was formed in the US called the "Affordable Heathcare Act"[3]. However, the website was poorly implemented and the site had many faults, which prevented citizens from signing up to or logging into the website [Connor 2013]. This affected the trust people had in ObamaCare, which was meant to be easy to sign up to. The website's failures were used to attack the new US heath care act the website was meant to help implement [Dann 2013]. As the website's problems were used to damage the act, Obama's political relations were damaged too, with Obama commenting "And let's admit it, with the website not working as well as it needs to work, that makes a lot of supporters nervous" [Connor 2013]. The cost of fixing the faulty website was reported to be $121m [FoxNews 2014]. The use of defect prediction could have allowed the developers to find defects before the system was released to the public, which could have saved ObamaCare from public ridicule and the extra cost of fixing the faulty website.

[1] The same could be said of a illness in a human. Machiavelli said that in the beginning of the malady it is easy to cure but difficult to detect, but in the course of time, not having been either detected or treated in the beginning, it becomes easy to detect but difficult to cure. [Machiavelli 1997]

[2] www.healthcare.gov

[3] The Affordable Care Act was implemented to put US citizens in charge of their health care. It is meant to give the American people a stable and flexible way of managing the choices available about their health.

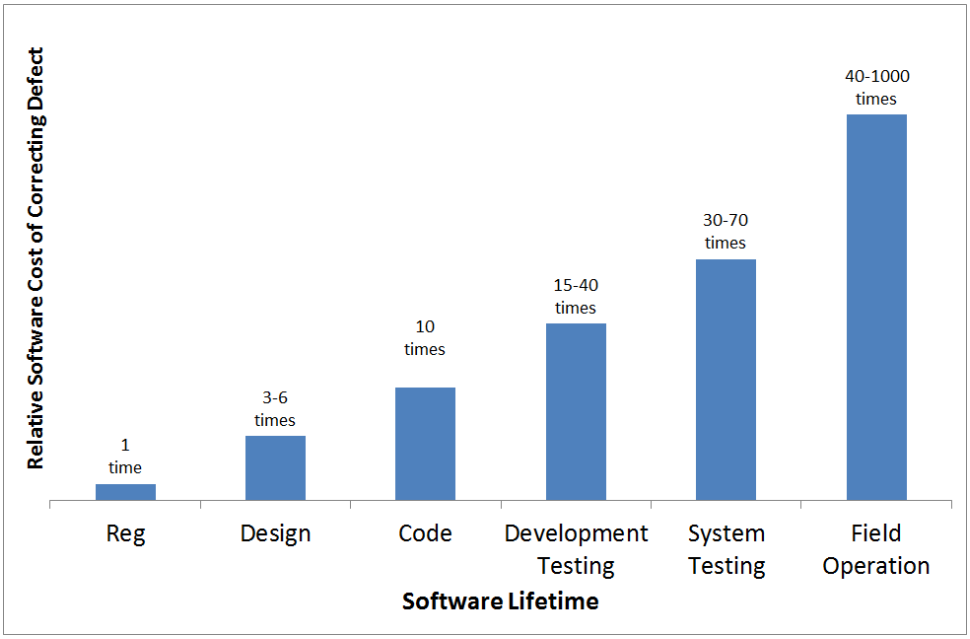


Figure 2.2: A bar chart showing the costs of defects within a software system at different stages of development. The costs rise due to the extra costs of fixing a defect at different stages of the development. Shown in Pressman [2001].

2.1.1 Performing Software Defect Prediction

Software defect prediction using machine learning is an automated method of determining potentially defective areas in a particular piece of software code. The predictions could make it possible for the developer to focus on areas of the software system before release, reducing the time and effort of finding defects by other means. Software defect prediction relies on three main components; dependent variables, independent variables and a model. Dependent variables are the defect data for the particular module or code. The defect data can be binary, or it can be continuous. Section 2.1.2 discusses dependent variables and how they are extracted in more detail. Independent variables are the metrics which can describe the software code, how it has changed or who changed it. Independent variables come in two forms, software code metrics; those that can be derived from the software code itself, and process metrics; metrics that measure the change of software code or software practices over time. Independent variables are discussed in more detail in Section 2.1.3. The model contains the rule(s) or algorithm(s) that predict the dependent variable from the independent variables. These rules can be as simple as the number of independent variables in the model, or be as complicated as decision trees[4] and regression[5] techniques. Figure 2.3 shows a diagrammatic representation of how defect prediction is carried out. Test and training data is made up of either dependent or independent variables. The training data is used to create a classifier. A classifier is an algorithm that is used to create the model that will be used to predict potential faults. This model is then tested by predicting where potential faults are in a system using the independent variables. To determine if these predictions are correct, the dependent variables from the test data are used. This will determine the performance of the model, by being able to calculate certain performance measures. There are many machine learning models currently being used to predict the

[4] A decision tree algorithm is one that creates a graph of decisions based on the chance of an event happening.
[5] Regression analysis seeks to determine best fit of independent value(s) based on a dependent value(s).

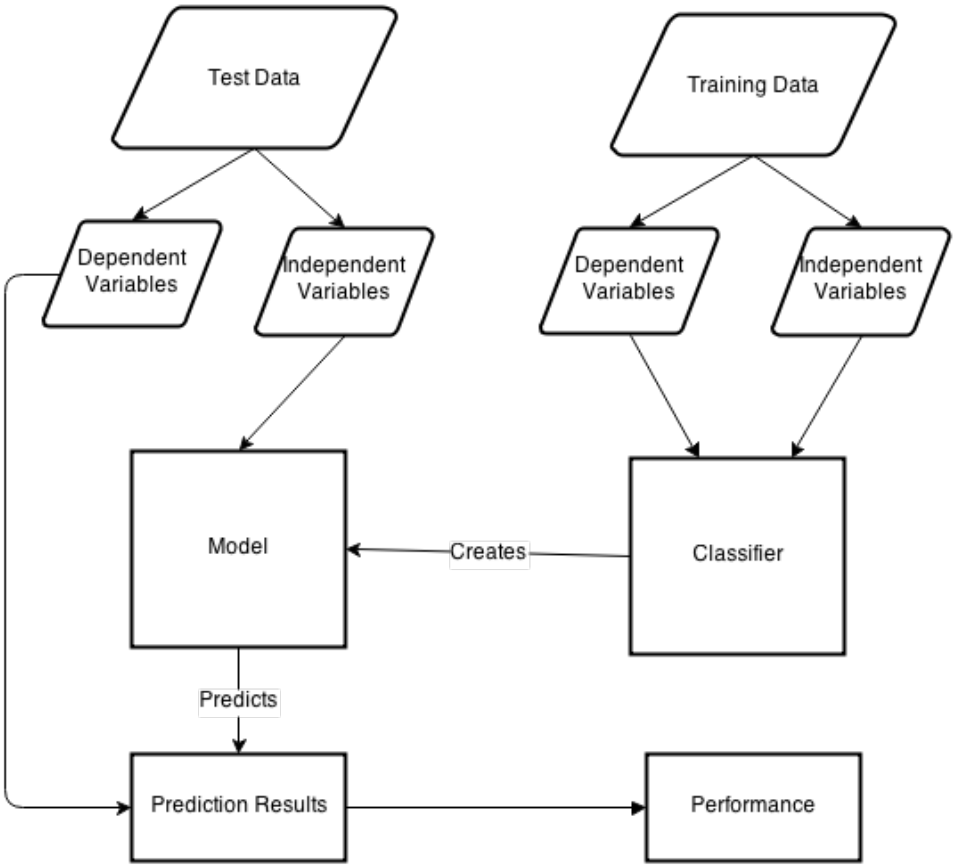


Figure 2.3: A diagram showing how defect prediction can be carried out.

location of defects in software systems. A study by Hall et al. [2012] has identified over 200 papers and the models/metrics used to carry out defect prediction.

2.1.2 Dependent Variables (Defective or Not?)

Dependent variables in defect prediction are the variables that indicate if a module is defective or not. The dependent variables in defect prediction can be categorical (i.e. a module is defective or not) or continuous (i.e. the number of faults in a module). A defect can come in two forms - a pre release or a post release defect. A pre release defect is one that is found by programmers during the development or by testers in the testing phase of development [IEEE 2010]. A post release defect is also known as a fault. A post release defect will only manifest itself as a fault when a user experiences a failure with the product [IEEE 2010]. This means that a defect could lie dormant within a software product. Section 2.1.2.1 below gives an example of the life cycle of both types of defect.

2.1.2.1 A Defect's 'Life'

A defect normally follows the same life cycle [IEEE 2010]. This life cycle can be used to determine the defectiveness of a module. An example of the 'lives' of two defects, one pre release and one post

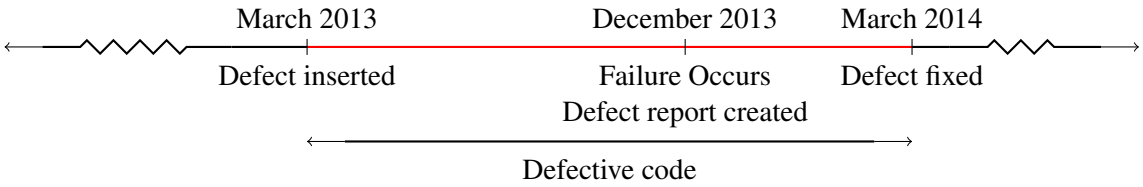


Figure 2.4: **A Defects's 'Life'** - this timeline shows the life of a defect within a software system. The defect was introduced in March 2013 and is not fixed until a year later. During this time, the code that is affected by this defect is labeled as defective (the red line).

release, is described below.

Thierry is a software developer at Zero Programming. In March 2013, he is tasked with developing a new feature for their popular tax reporting software. Unfortunately, Thierry unintentionally programs two defects (defect A and B) into one of the methods that he created. Defect A is found during the testing phase of development and is fixed by Thierry and is therefore recorded as a pre release defect. However, during the testing phase Defect B passes the current tests and is compiled into the final product. Defect B lies dormant for around nine months. In December 2013, a user experiences a failure from the software due to Defect B and files a defect report. This defect report is added to Zero Programming's defect report database and means that Defect B has become a post release defect or a fault. Another developer, Patrick, is assigned the task of fixing the fault. A full year later after Defect B was introduced, the fault is fixed by Patrick and he files the defect report as fixed.

Figure 2.4 shows the timeline of Defect B which was present in Zero Programming's tax software system. The use of this timeline allows the identification of methods which are defective at any stage of the products development. If the insertion and fix point dates are known then at any point in between these two dates (the red line on Figure 2.4), the method will have been defective. If one knew the method introduced by Thierry in the example above is defective in November 2013 then the defect insertion and fix dates are known. The insertion and fix points of a defect are found in this dissertation by using the algorithm described by Śliwerski et al. [2005]. This algorithm is described in more detail in Chapter 4.

Dependent variables can be predicted by a model to forecast if a module is defective or not. This result can then be tested to see if the forecast is correct or not. This testing can help determine the recall and precision of the model. Precision and recall are measures of relevance of the data used. Precision is a measure of the accuracy of the model used to predict defects (i.e of all the instances predicted defective, how many are actually defective). Recall is the measure of relevant retrieval of instances (i.e. how many instances are identified by the model as defective out of all those defective instances that should have been returned).

Precision is calculated as follows (where D = number of defects):

$$precision = \frac{|D_{relevant} \cap D_{found}|}{|D_{found}|} \tag{2.1}$$

and recall is calculated as:

$$recall = \frac{|D_{relevant} \cap D_{found}|}{|D_{relevant}|} \tag{2.2}$$

2.1.3 Traditional Independent Variables (Software Metrics)

There are many different traditional independent variables that are used within defect prediction [Hall et al. 2012]. Sections 2.1.3.1 and 2.1.3.2 detail these independent variables, describing which have been used, why they were used and how effective they have been.

2.1.3.1 Source Code Metrics

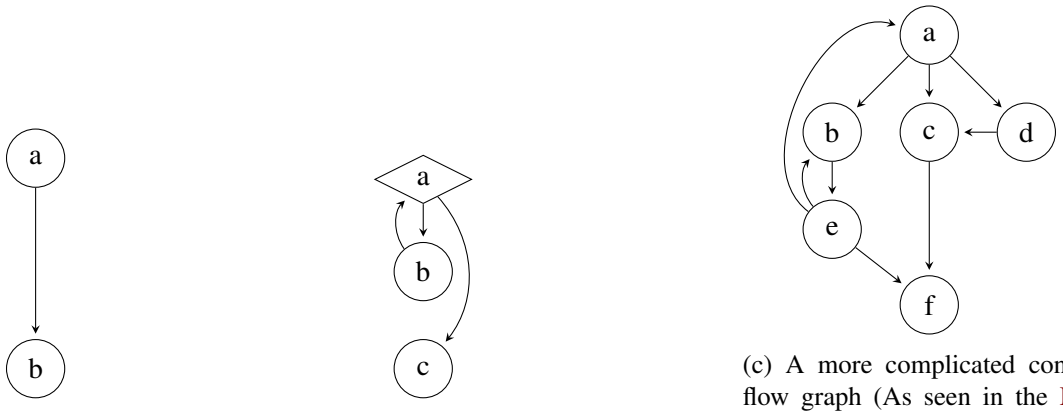
Previous work on features of code in relation to defects has focused on defining and evaluating source code metrics (SCM). SCM measure particular code features (e.g. lines of code, object oriented metrics and complexity metrics). These metrics and the studies they appear in are described below.

Lines of Code LOC was introduced as a simple measure that allows the use of size to indicate potential defective areas of a software system. Some of the advantages of using LOC is that LOC are very quick to calculate and easily transferred across programming languages. One of the first defect prediction models proposed by Akiyama [1971] was based on lines of code (LOC). Akiyama [1971] proposed a regression model for determining the number of defects in the code based on the LOC. LOC measure the lines of code in a particular module, file or package. Studies differ in their interpretations of a LOC, some studies will use code comments, some blank lines and others will omit these lines. Others use logical lines of code, which is where only the semi colons have been counted [Rosenberg 1997]. However, Rosenberg [1997] highlighted that the format of the LOC was irrelevant as they all correlate with one another.

Fenton and Ohlsson [2000] analysed pre and post release defects of a large communications system. They found that LOC was good at ranking the most fault-prone modules. The results of the Fenton and Ohlsson [2000] study have been replicated and confirmed by Andersson and Runeson [2007] and Galinac Grbac et al. [2013]. Zhang [2009] confirmed Fenton and Ohlsson [2000]'s results that there is a relationship between the LOC and defects at both package and file level. Ostrand et al. [2005] produced a simple LOC based model to predict defects in a large industrial system. They concluded that their LOC model was a good way of predicting defects, with the model finding around 70% of faults. Bell et al. [2006] used the Ostrand et al. [2005] model on a different software system, an automated voice response system. In this study Bell et al. [2006] found that the LOC model was not as effective, only finding 55% of the defects. Gyimothy et al. [2005] found that LOC was a very significant indicator of defects by performing regression and machine learning analysis on the web program Mozilla[6]. Subramanyam and Krishnan [2003] produced the same result when they analysed a commercial Java/C++ system.

LOC predicts defects with an average recall of around 70% and an average precision of 67%, across 17 studies [Hall et al. 2012]. A problem with LOC is that it measures only one coarse grained feature of code and may only provide limited insight into potential sources of defects. Measuring just the size of a module does not take into account the finer detail involved, for example how complex the code is or how the code interacts with the system. Collinearity between the size of the module and the defect density exists in LOC [Rosenberg 1997]. This makes some findings using LOC confirm a statistical property, rather than actually show a relationship between the size of a module and the defectiveness chance of that module.

[6] http://www.mozilla.org/



(a) A simple control graph. v(G) = 1 - 2 + 2.

(b) A while loop control flow graph. v(G) = 3 - 3 + 2 = 2

(c) A more complicated control flow graph (As seen in the McCabe [1976] paper). v(G) = 9 - 6 + 2 = 5

Figure 2.5: Examples of control graphs and their calculated complexity scores

Complexity Measures Complexity measures were introduced to provide a measure of how hard software code may be to comprehend, test and maintain. The measures were formed in order to gain measurable properties of the software and the relationships between these properties [Halstead 1977]. Halstead metrics [Halstead 1977] and McCabe's complexity measure [McCabe 1976] were two of the first complexity measures introduced.

Cyclomatic complexity (CC) is based on the number of decisions in a program with McCabe [1976] and Halstead [1977] having their own measures. CC measures the human comprehension of the code by identifying branching structures in code and calculating the number of logical paths through the code. The higher the CC number, the more complex the code and therefore the higher the probability of a defect being present.

To calculate the CC value you have to create a control flow graph of a particular module. A control flow graph is a representation of all paths of a module that could be traversed through a program during its execution. Figure 2.5 shows some examples of control flow graphs. Certain components of the control flow graphs are used to calculate CC. The number of nodes in a system (N). The number of times control is passed from one node to another, an edge(E). Finally, the exit node or the number of disconnected parts of the flow graph (P). Equation 2.3 shows how these components are used to calculate the CC (v(G)) of any control flow graph (G).

$$v(G) = E - N + 2P \tag{2.3}$$

Halstead [1977] created a set of metrics that aimed to provide insight into code complexity and developer effort. The metrics are based on four base measures:

1. Number of unique operators[7] - n_1
2. Number of unique operands[8] - n_2
3. Total number of operators - N_1

[7] Operators include assignments, blocks, labels, if statements, while statements and statement terminations (i.e. in Java ;).
[8] An operand is a subroutine declaration or variable declaration.

Metric	Equation	Description
Length	$N = N_1 + N_2$	Sum of all operators and all operands. A size metric that is an alternative to LOC.
Vocabulary	$n = n_1 + n_2$	Sum of all unique operators and operands. High values indicate harder to read code and therefore difficult to maintain.
Volume	$V = N.log_2(n)$	Another size metric. Describes the content in bits.
Difficulty	$D = \frac{n_1}{2} \times \frac{N_2}{n_2}$	Measures how difficult the code is to write, thus how error prone it may be.
Level	$L = \frac{1}{D}$	A low level score indicates less error prone code.
Effort	$E = D \times V$	Measures the effort to understand the code. The higher the metric the more difficult the code is to maintain.
Content	$C = L \times V$	Language independent complexity measure
Error Estimate	$B = \frac{V}{300}$	This metric aims to predict the number of validation bugs. 300 is the proportion of defects within the system.
Programming Time (secs)	$T = \frac{E}{18}$	How long it would take to program a particular module. 18 is a constant that reflects the number of decisions a programmer will have to make per second.

Table 2.1: Halstead complexity metrics created by Halstead [1977].

4. Total number of operands - N_2

These four measures form the basis of the algorithms shown in Table 2.1. The CC measures have been heavily scrutinised due to the fact they assume certain constants (error estimate and programming time) and may only be a proxy for LOC [Fenton and Pfleeger 1998, Hamer and Frewin 1982, Shen et al. 1983, Shepperd and Ince 1994]. CC metrics again also only provide a subset of potential features that may be associated with the faultiness of potential code.

Ohlsson and Alberg [1996] carried out an investigation using a telecommunications system with the aim to study the relationship between defects and several graph metrics, including but not limited to, the number of branching points, number of branches and the number of possible paths. Ohlsson and Alberg [1996] also calculated McCabe's complexity and wanted to predict which modules could be faulty before coding had already begun. Ohlsson and Alberg [1996]'s results showed that the metrics could predict the most fault prone modules before coding had started. Turhan et al. [2009] used CC alongside many other metrics to create a defect prediction approach for companies that did not track local bug data. Menzies et al. [2007] compared the use of metrics and models within defect prediction. Using the McCabe metric data taken from the NASA datasets Menzies et al. [2007] were able to show that the SCM selected is not as important as which learning algorithm is used.

Object Oriented Metrics Following the introduction of new object oriented programming languages, researchers began to take advantage of the new metrics they could derive from the interactions between

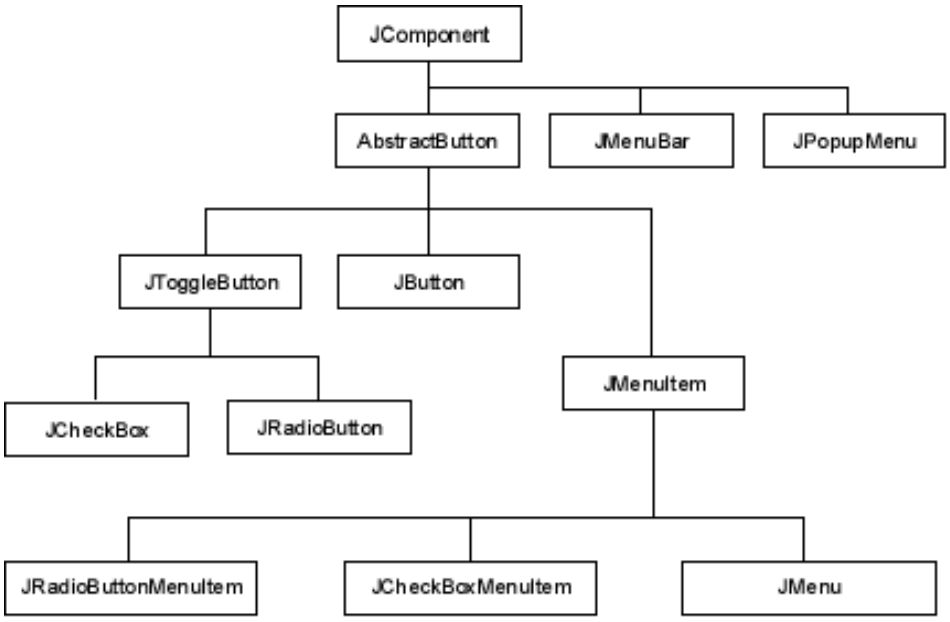


Figure 2.6: Inheritance between Swing classes in Java.

and within objects. Chidamber and Kemerer [1994] (CK) developed six Object Oriented (OO) metrics to measure the object oriented features of code. These six OO metrics focused on the complexity of classes within a system and how complexity can impact on maintenance and development. The metrics are based on the notion that the higher the complexity of a method and/or its class, the higher the potential for errors.

The CK metrics use the notion of inheritance within the code and mention trees, parents and children. These terms will be explained briefly. Inheritance is where a subclass (child) `Y` inherits all of the attributes and operations associated with its superclass (parent) `X` [Pressman 2001]. Inheritance is an important part of object oriented programming as it allows code reuse. An inheritance tree is similar to a family tree in that it is the full structure of how all the parents and children of a system are linked. Figure 2.6 shows an example of an inheritance tree. Each of the white rectangle boxes are an object in Java swing package. Each branch that flows down from an object is that objects children. The children inherit methods and attributes from its parent object. For example, a `JMenu` will inherit methods from its parent `JMenuItem`. This is just a portion of the swing inheritance tree and `JComponent` will have a parent and `JMenu` could have children.

The six OO metrics described are outlined below [Chidamber and Kemerer 1994]:

1. **Weighted Methods per Class (WMC)** - WMC is the sum of the complexities for all the methods in a class. The number of methods and the complexity of these methods is a predictor of the amount of maintenance needed for the class. The more methods in the class, the greater the impact on the children as the children inherit all the methods in that class. Classes with many methods are more likely to be application specific, limiting their reuse potential.

2. **Depth of Inheritance Tree (DIT)** - The DIT metric is the maximum length from the node to the root of the tree. DIT measures the amount of ancestor classes that can affect the class analysed.

Metric	Description
FanIn	Number of other classes that reference the class
FanOut	Number of other classes references by the class
NOA	Number of attributes
NOPA	Number of public attributes
NOPRA	Number of private attributes
NOAI	Number of attributes inherited
NOM	Number of methods
NOPM	Number of public methods
NOPRM	Number of private methods
NOMI	Number of methods inherited

Table 2.2: Class level OO metrics described in the defect prediction literature [D'Ambros et al. 2010]

The deeper the inheritance tree, the greater the number of methods that have been inherited. This makes it more difficult to predict behaviour. Deep trees mean there is a higher design complexity.

3. **Number of Children (NOC)** - NOC is the number of immediate subclasses subordinated to a class in the class hierarchy. It measures how many subclasses are to inherit the methods of the parent class. The greater the number of children, the greater the reuse, the greater the likelihood of improper abstraction of the parent class. If a large number of children exist, this could be a case of misuse of subclassing.

4. **Coupling Between Object classes (CBO)** - CBO is how many classes a single class uses. One class is coupled to another if it acts on the other. Generally the greater the coupling, the more detrimental to design as it prevents reuse. If there is low coupling, the class can be reused elsewhere. If there is a large amount of couples, then maintenance becomes bigger as the changes done in one class will effect other parts of the system.

5. **Response For a Class (RFC)** - This measures the amount of methods that can be executed in response to a message received by an object in that class. It also measures the potential communication between the class and other classes. The greater the amount of methods that can be invoked in response to an object, the greater amount of understanding required from the tester or developer. This means that the testing and debugging of the class is more complicated.

6. **Lack of Cohesion in Methods (LCOM)** - LCOM measures the count of the number of method pairs whose similarity is zero minus the count of method pairs whose similarity is not zero within a class. The larger the amount of similar methods, the more cohesive the class. Cohesiveness of methods in a class is desirable as it promotes encapsulation. A lack of cohesion implies that the class should be split into two or more subclasses. Low cohesion also increases complexity, thereby increasing the likelihood of errors.

Table 2.2 shows the other OO metrics that have been identified in the defect prediction literature [D'Ambros et al. 2010].

Basili et al. [1996] investigated OO metrics to see how effective they were as predictors. They investigated eight medium sized information management systems written in C++. Basili et al. [1996]'s conclusions were that five out of the six CK metrics were useful predictors during the early phase of

development. The only predictor that was not good was the WMC metric [Basili et al. 1996]. Similar types of analysis have been performed by Briand et al. [1999], Chidamber et al. [1998], Janes et al. [2006], Li and Henry [1993], Ronchetti et al. [2006]. Each of the studies is performed on industrial C++ projects, except for Li and Henry [1993] which was done in Ada. Each of the studies conclude that at least one or more of the metrics is good at predicting defects. El Emam et al. [2001] used metrics from McCabe [1976] and the metrics from the Briand et al. [1999] study to investigate different defect prediction models. El Emam et al. [2001] results showed that their model had high accuracy and that coupling had the strongest association with fault proneness. Briand et al. [1999] gathered their metrics from a commercial Java application. Briand et al. [1999]'s work showed that the coupling metrics had the strongest association with fault proneness.

Hall et al. [2012] show that in defect prediction OO metrics have a similar recall and precision as LOC, in 42 studies the average recall is around 65% and precision around 67%. Compared to LOC, the OO metrics do measure some finer grained features of code and also identify more of those features. However, these OO metrics are still only a small subset of possible code features.

Combination of Metrics Some studies have used a combination of SCM to generate their models. Arisholm and Briand [2006] used 32 different SCM in their defect prediction model. These metrics included those measuring the coupling between classes, the changes to code (e.g. fault corrections) and code quality (code style, practices and the amount of redundant code). They used the model on a large Java telecommunications system which has over 110K SLOC. Arisholm and Briand [2006]'s results showed that their model could reduce verification effort costs[9] by up to 29%. Khoshgoftaar and Seliya [2004] used metrics based on "call graphs, control flows and statement metrics". In total 28 metrics were used to compare seven different classification models, examples of these metrics included: procedure calls (call graphs), number of entry nodes (control flows) and LOC (statement metrics). Khoshgoftaar and Seliya [2004]'s results showed that software quality estimation models can improve the reliability of a software system, due to the models being able to target specific potentially defective areas of the code base.

2.1.3.2 Process Metrics

A process metric reflects the changes of a system over time [Henderson-Sellers and Henderson-Sellers 1996] (e.g. the number of code changes in a module). Process metrics are based on revisions and historic changes to files. They can take into account the amount of times the file or module has been changed due to defective code.

The histories of the revision control systems have been used to indicate defects in software systems. Table 2.3 shows an example of the process metrics introduced by Moser et al. [2008] based on the historical data from the revision control system. Moser et al. [2008] investigated whether change metrics are better predictors of defects than code metrics. They also wanted to analyse the cost associated with getting a prediction incorrect. They compared 31 code metrics to 18 change metrics over three different releases of Eclipse (2.0, 2.1 and 3.0). Moser et al. [2008] concluded that change metrics are a better predictor of defective code than code metrics. They also showed that files with a high number of revisions and files with bug fixing activities are the best indicators of potential defects. Whilst, heavily edited files or files included in a large CVS commit are less likely to be defective.

[9] Costs incurred from programmers checking whether a potentially defective module is actually defective.

Metric Name	Definition
REVISIONS	Number of revisions of a file
REFACTORINGS	Number of times a file has been refactored
BUGFIXES	Number of times a file was involved in bug-fixing
AUTHORS	Number of distinct authors that have committed the file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
AVG_LOC_ADDED	Average number of lines of code added for all revisions
LOC_DELETED	Sum over all revisions of the lines of code deleted from a file
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVG_LOC_DELETED	Average number of lines of code deleted for all revisions
CODECHURN	Sum of (added lines of code - deleted lines of code) over all revisions
MAX_CODECHURN	Maximum CODECHURN for all revisions
AVG_CODECHURN	Average CODECHURN for all revisions
MAX_CHANGESET	Maximum number of files committed together to the repository
AVG_CHANGESET	Average number of files committed together to the repository
AGE	Age of the file in weeks
WEIGHTED_AGE	Sum of (the Age of revision times the LOC added) divided by the sum of the LOC added

Table 2.3: List of Change metrics used in the Moser et al. [2008] study.

Graves et al. [2000] examined a telephone switching system which consisted of over 1.5 million LOC. Graves et al. [2000] approach was based on seven new measures derived from the revision history - Number of past faults, Number of deltas (i.e. the amount of previous changes), the average age of the code, the development organisation, number of developers, how modules are connected (i.e. how many modules are changed together) and a weighted time damp model. The weighted time damp model computes a module's fault potential by adding contributions from each change. A contribution is the level of fault potential, a large fault potential is computed if the change is recent and large. Graves et al. [2000] concluded that the sum of the contributions (the weighted time damp) was the best predictor of faults in a system. They also concluded that the number of developers has no influence on the defect potential, or the extent to which a module is connected with another module.

Ostrand et al. [2005] created a model based on revision history information. The information selected for the model was based on characteristics that had been associated with high levels of faults (e.g. the change status of the file, file age and number of faults identified in the previous release). This model was used on 15 different releases of a large industrial system. The model proved to be very efficient with the top 20% of the files identified as most defective containing at least 84% of the faults [Ostrand et al. 2005]. The model was used again to investigate an automated voice system by Bell et al. [2006] and attained similar results to the previous study. Weyuker et al. [2006] attempted to generalise the model described by Ostrand et al. [2005] to be able to apply to different systems without extensive statistical expertise or modeling effort. Hassan and Holt [2005]'s method also uses bug data to create a prediction model. They created a top-ten list approach which validated heuristics about the defect proneness of the most recently changed areas of code and fixed defects. Hassan and Holt [2005] concluded that the most recently fixed code is also the most defect prone. Bernstein et al. [2007] used the defect change information from six different Eclipse plug-ins to create non-linear prediction models.

Ratzinger et al. [2007] used the information from historical revisions to create three different defect prediction models. Ratzinger et al. [2007] applied these models to five different industrial projects. Ratzinger et al. [2007]'s models used 63 different features extracted from the revision histories and bug tracking systems. Ratzinger et al. [2007] concluded that their models can accurately predict short term defects but the accuracy of the model is affected by the severity of the defect. The prediction of defects with higher severity has lower precision than those with lower severity.

Code churn (i.e. the number of modified lines in a file or module per commit) has been researched extensively by a number of researchers [Hassan and Holt 2005, Hassan 2009, Illes-Seifert and Paech 2010, Khoshgoftaar et al. 1996, Kim et al. 2007, Layman et al. 2008, Nagappan et al. 2010, Nagappan and Ball 2005, Nikora and Munson 2003, Purushothaman and Perry 2005, Schröter et al. 2006, Śliwerski et al. 2005]. Khoshgoftaar et al. [1996] used code churn to identify defective modules in two large legacy communications systems. Modules were classed as defect prone depending on the amount of lines that were added, deleted or edited in a module during the debug phase of development. Four code churn metrics were introduced by Layman et al. [2008] that depend on modified blocks. A block is "a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end" [Aho et al. 1986]. Table 2.4 shows these metrics in more detail. Layman et al. [2008] used these metrics to create a model based on stepwise logistic regression. Layman et al. [2008] collected churn metrics from snapshots of a Microsoft program with around one million LOC. The precision of Layman et al. [2008]'s models was around 73%-86%. The metrics in the Layman et al. [2008] paper were also used to investigate further Microsoft products. Nagappan and Ball [2005] showed that relative churn is better than churn at influencing defect density in Windows Server 2003. Schröter et al. [2006] examined 52 Eclipse plug-ins and used information of past post-release failures to predict future failures at both file and package level. This information was based on which packages were used within the developed files or packages. Schröter et al. [2006]'s results showed that information of specific use of packages in one failed file/package could be used to predict future failures in another file/package.

Change bursts, "a sequence of consecutive changes", have been investigated as predictors of defects. Change bursts look at code churn over a set number of days with a specified gap size. Nagappan et al. [2010] investigated the binaries of 3,404 different versions of Windows Vista - each consisting of over 50 million LOC. Change bursts were shown to have "excellent predictive power" with precision and recall exceeding 90% [Nagappan et al. 2010]. Illes-Seifert and Paech [2010] investigated a number of different process metrics and concluded that "frequency of change" was a very good indicator for predicting defects. Purushothaman and Perry [2005] investigated small changes in the code. They showed that small changes to code were unlikely to result in a error in the module (a one line change has less than 4% probability of causing a defect). They also showed that nearly 40% of the changes made to correct code introduced a defect into the software [Purushothaman and Perry 2005]. This finding was also observed by Śliwerski et al. [2005]. The authors found that it is three times more likely that a fix in an Eclipse project induces a further fix in the future compared to an enhancement. The authors also showed that the larger the change to a file, the more likely that change is going to need fixing in the future [Śliwerski et al. 2005]. The entropy of a change was investigated by Hassan [2009]. The entropy of a change is the complexity associated with the code changes (i.e. the size of the modification of a change). The premise being that a change that involves many different changes will be harder to undertake, than a change that is smaller. The entropy metric, when compared against the amount of changes and the amount of previous bugs across six different open source systems, was better at predicting where defects may occur [Hassan 2009].

Churn Metric	Definition
Churn	*New blocks(new) + Changed blocks(new)*
Relative Churn	$\frac{\textit{New blocks(new) + Changed blocks(new)}}{\textit{Total blocks(new)}}$
Relative Churn	$\frac{\textit{Deleted Blocks(new)}}{\textit{Total blocks(new)}}$
NCD Churn	$\frac{\textit{New blocks(new) + Changed blocks(new)}}{\textit{Deleted Blocks(old)}}$

Table 2.4: The four churn metrics based on code blocks introduced by Layman et al. [2008]

Nikora and Munson [2003] used code churn, but instead of the lines, they used the churn of software metrics. The authors measured the difference in the control flow characteristics of the source code.

Hassan and Holt [2005] wanted to show managers the top 10 subsystems that are likely to have problems using a cache system. Using a cache of the most frequently/recently modified files and the most frequently/recently fixed files to show the top 10 files that are susceptible to a fault. These lists are able to dynamically update to reflect the current risks and be clear and easy for managers to understand. This method is only good for short term prediction and the results were not that great, having only a recall rate of over 60% for some of the lists [Hassan and Holt 2005]. Kim et al. [2007] followed on from the Hassan and Holt [2005] work. Kim et al. [2007] wanted to show that bugs occur in bursts of related faults. Kim et al. [2007] analysed seven different software systems and used a cache to hold the locations of the most fault prone entities. BugCache held the locations of the last known faults and FixCache held the locations of where the bug has been fixed. Kim et al. [2007]'s reasoning being that when a fault is fixed in a location, there is a high chance that a bug will appear there in the future.

2.1.4 The Problem with Traditional Independent Variables

Most of the traditional independent variables (as described above) have been extensively used in previous defect prediction studies [Hall et al. 2012]. Despite all this research, Menzies et al. [2010] reported that better mining technology is not leading to better defect prediction models. The current models have reached a predictive limit and a new approach is needed.

One possible reason for the limit could be that the current metrics suffer from being very coarse grained and only have the capability to measure only a small sub-set of code features. Gray et al. [2011] suggest that the coarse grained nature of such metrics prevents machine learning techniques from effectively differentiating between defective and non-defective modules: if one module has the same metric values as another (say in terms of LOC), but they have not been labelled the same in terms of their defectiveness, this will hinder the learning algorithm's ability to learn. Gray et al. [2011] identifies many modules in the NASA datasets which have identical values across a range of metrics but different defectiveness labels. This is a problem for the learning algorithm to predict accurately in two ways. The first is that the repeated instances may not actually be the same pieces of code but are labeled the same. The second is that the repeated instances may have the same metrics, but are not labeled as defective the same. Current metrics may not be picking out the differences between different code or that the identical instances are not being consistently/correctly labeled. This could mean that the results that have been made using

these data sets could be finding suboptimal results [Gray et al. 2011]. Kim et al. [2011] show that the predictive power of a technique is impacted by the amount of noise in the data set. Kim et al. [2011] suggests that the current commonly used set of metrics are not sufficient to differentiate modules for defect prediction.

2.1.5 Current Finer Grained Techniques

Some defect prediction studies have used less traditional independent variables and have focused on finer grained details of the source code. These independent variables have mostly been based on analysing the text of the code. One such approach is the use of metrics based on a lexical analysis of text in the code [Zeller et al. 2011]. Zeller et al. [2011] analysed the source code of Eclipse 2.0, 2.1 and 3.0 for the number of 8-bit ASCII characters in a file. Even though their experiment was carried out to "demonstrate a number of blunders in research" the method was shown to have reasonable results. Binkley et al. [2007; 2009] use an information retrieval based defect prediction technique, known as the QALP score to help identify potential defects. The QALP score measures the similarity between a module's comments and its source code using a cosine similarity. The results showed that, when used alongside LOC, the QALP score is helpful in predicting defects in files. Marcus et al. [2008] also proposed a method based on an information retrieval technique - latent semantic indexing (LSI). They used LSI to analyse the text of source code to develop a new measure for class cohesion called C3. C3 measured the "how strongly the methods of a class related to each other". Marcus et al. [2008] were able to use this technique alongside previous structural metrics to attain better results for defect prediction than with structural metrics alone. Mizuno et al. [2007] used spam filtering techniques to create a defect detection technique. Mizuno et al. [2007]'s technique treated source code as text files and used the text mining techniques used in spam filtering to identify problematic patterns in the text. The technique was able to classify more than 75% of modules correctly. Abebe et al. [2012] improved the detection of defects by using lexicon bad smells (LBS) in conjunction with software structure metrics. Examples of lexicon bad smells are short terms used as identifiers (e.g abbreviation or acronym) and meaningless terms (e.g. `foo` and `bar`). Poor quality lexicon for identifiers has been shown to be associated with the introduction of faults [Butler et al. 2009]. Arnaoudova et al. [2010] investigated the link between identifier terms and fault proneness. Arnaoudova et al. [2010]'s motivation was that identifier terms used in differing contexts (i.e. the same identifier terms used in different parts of a system but with never the same meaning) decreased code comprehension. Arnaoudova et al. [2010] measured the entropy (how dispersed the identifier terms are within a software system) of identifier terms within certain contexts; the higher the entropy, the more scattered the identifer term is within the system. Arnaoudova et al. [2010]'s initial results showed that if a method has identifer terms with high entropy, the more chance that method has of being fault prone.

Petric and Grbac [2014] investigated finer grained software code structure in an attempt to identify defects. Petric and Grbac [2014] identify the code structure as intra class subgraphs called. Petric and Grbac [2014] limit the size of these subgraphs to three node connected subgraphs. Figure 2.7 shows the 13 subgraphs studied. Each of the lines dots on these is a node in a network and the lines (edges) are how these nodes are connected. The ids for the subgraphs are located above each subgraph. Petric and Grbac [2014]were able to find associations between certain subgraphs and defects. Shivaji et al. [2009] investigate the features of code. Shivaji et al. [2009] use a modified version of the bag-of-words approach (BOW+ [Scott and Matwin 1999]) to identify a sub-set of code features based on operators such as !=, ++ and &&. Overall these previous approaches have shown promise in terms of defect prediction but have not been fully exploited.

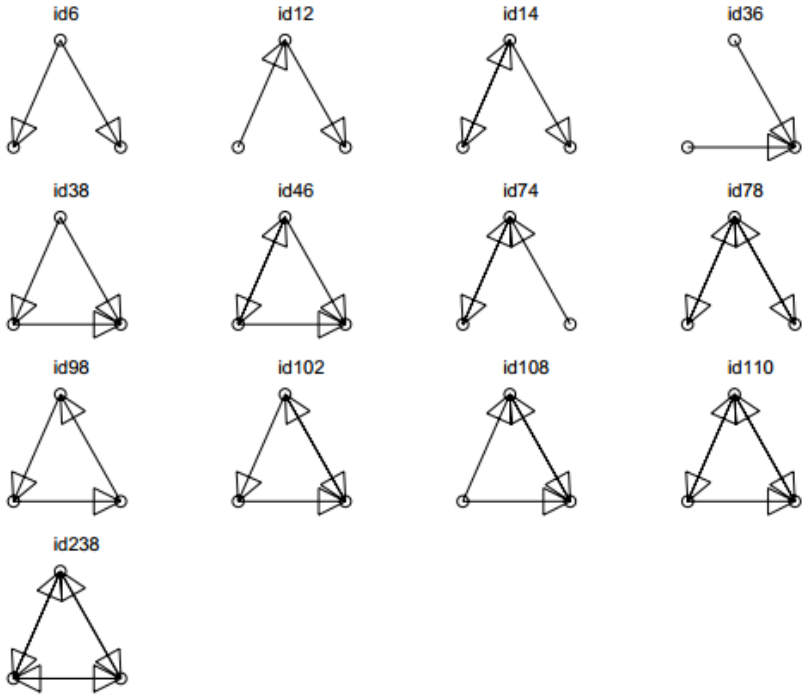


Figure 2.7: The three node subgraphs examined by Petric and Grbac [2014].

2.2 An Alternate View of Code Motivated by DNA Fingerprinting

Defects are not limited to software code. In biology, the DNA (code) of living organisms can become faulty and these faults bring about genetic disorders. Biological scientists have developed approaches to detect these disorders. The approach discussed in this dissertation is motivated by how these genetic disorders have been identified in biology. The approach taken in biology has been adapted for use within Java software code.

2.2.1 Genetic Disorders and their Discovery

Genes carry the information to build a living organism [NationalInstituteofHealth 2014]. Sometimes there is a mutation in these genes and this mutation could cause the instructions to fail. This failure causes a genetic disorder [NationalInstituteofHealth 2014]. The mutation could happen when the genes are passed from parent to a child, or just naturally during a organism's lifetime. There are three types of genetic disorders [NationalInstituteofHealth 2014]:

1. Single-gene - a mutation affects one gene.
2. Chromosomal - the chromosome (or part of) which holds the gene is missing or changed.
3. Complex disorders - mutations are in two or more genes.

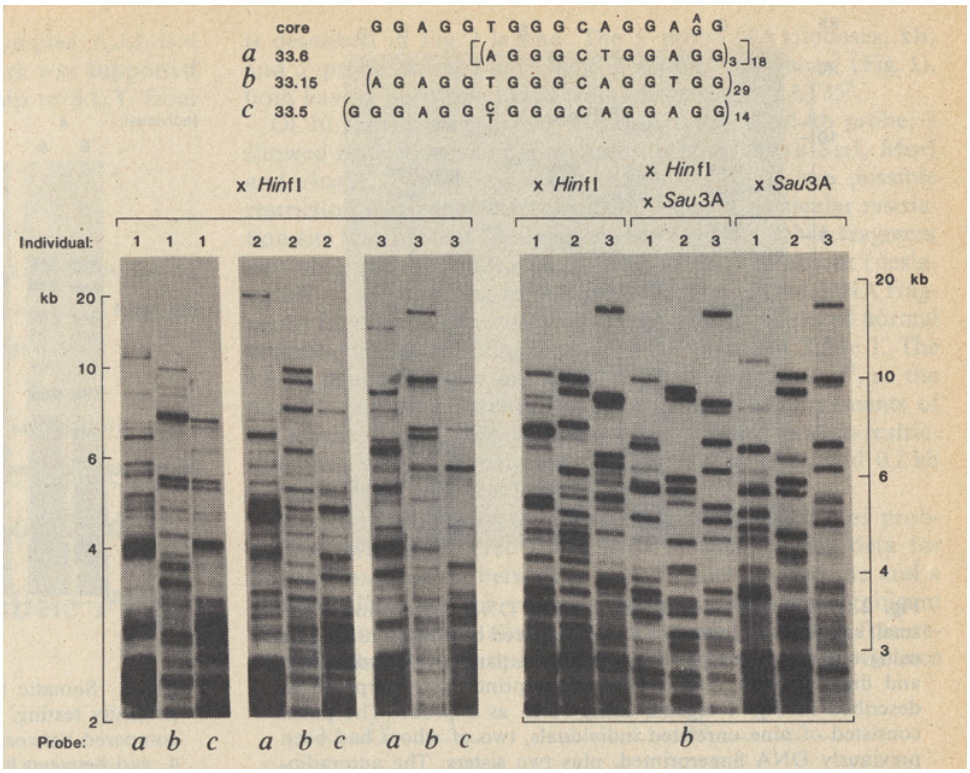


Figure 2.8: A example of a Southern Blot. Taken from Jeffreys et al. [1985] article in Nature

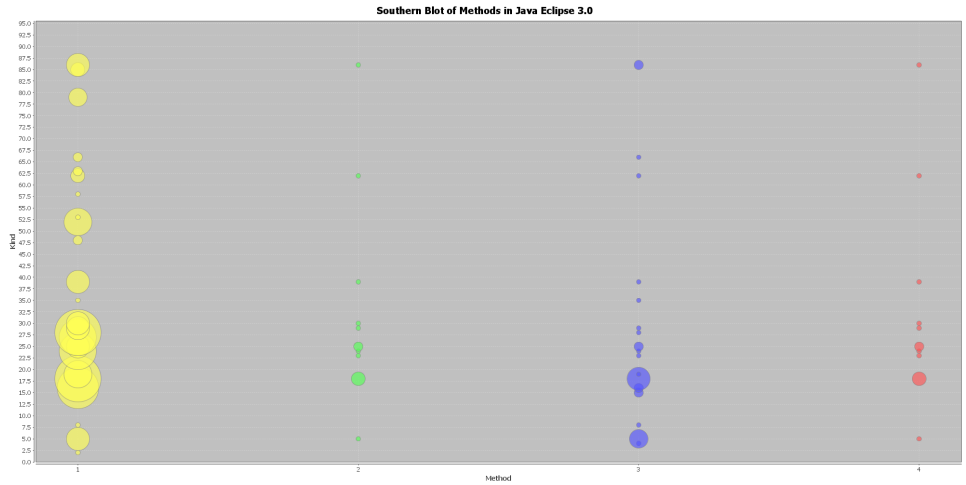


Figure 2.9: An interpretation of a southern blot for Java methods

Genetic testing is a medical test that identifies changes in chromosomes, genes or DNA [U.S.NLM 2014]. More than 1,000 tests are currently in use and more are in development [U.S.NLM 2014]. Scientists used to investigate genetic disorders (defects in the living organism) by manually observing chromosomes. A chromosome is *"an organised package of DNA found in the nucleus of a cell"* [Institute 2014]. Humans have 23 different pairs of chromosomes and these chromosomes are used to discover if the existence/non existence of a certain chromosome is associated with a particular genetic disorder. For

example, the genetic disorder Down Syndrome is caused by an extra or partially extra third copy of the chromosome 21[10] [Patterson 2009]. This information could be used to predict whether a foetus or new born baby has a genetic disorder. A chromosome has the sequence of DNA within it and it is this DNA sequence that allowed scientists to perform finer grained techniques to predict genetic disorders.

DNA contains the biological instructions that make each species unique. It carries instructions that are passed from the adult organism to the child during reproduction [Insitute 2011]. Each human's DNA is 99.9% the same as another humans, the only difference is small sections of the DNA which have different sequences of the four base pairs [Insitute 2011]. It is the order of these base pairs that determine the look and the behaviour of a human being. Every person can be identified solely by the sequence of their DNA; however there are so many millions of base pairs (estimated to be circa 3.0×10^9 base pairs) that the task would become very time consuming. Instead scientists use a shorter method to identify differences between two DNA sequences based on the repeating patterns in DNA [Brinton and Lieberman 1994]. DNA profiling uses repetitive sequences that have identified as being highly variable called variable number tandem repeats (VNTR). VNTR loci are very similar between closely related humans, but also so variable that unrelated individuals are extremely unlikely to have the same VNTRs.

Jeffreys et al. [1985] produced the first DNA fingerprint in 1984 by using probes based on tandem repeats of sequences to isolate 'minisatellite' regions [Jeffreys et al. 1985]. His technique is a form of Restriction Fragment Polymorphism (RFLP). RFLP is based on DNA probes. Jeffreys et al. [1985] discovered that it was possible to design probes for the cloning of individual polymorphic minisatellite regions from human DNA and that these can analyse multiple hypervariable regions. A probe is a core sequence repeated in tandem and provides a set of genetic markers for an individual. Jeffreys et al. [1985] used variable number tandem repeats (VNTRs) as his probes. VNTRs are a repeated sequence that has between 10 and 60 nucleotides. A person's DNA breaks up into millions of fragments when it undergoes restriction digestion. Using the probes Jeffreys et al. [1985] was able to identify a subset of restriction fragments containing the VNTR repeat. Longer restriction fragments migrate slower than larger ones when separated by electrophoresis[11]. A southern blot is created when the separated DNA is transferred to a solid membrane. When the probes are labeled with a radioactive isotope, phosphorus-32 (^{32}P), the probes bond to fragments containing the relevant VNTR. The final process result is exposed to x-ray film. The radioactive activity from the probes forms black bands on the film that corresponded to the positioning of the VNTR.

DNA sequencing has provided alternative gene testing techniques, making tests faster and more accurate [Karthikeyan 1999]. Before DNA sequencing, testing was limited to examining chromosomes and genes. This was a very coarse grained method, as there are only 23 chromosome pairs to examine, it was able to determine some gene problems, but not all. DNA, with its sequence of millions of base pairs, is much finer grained and contains a lot more information about the instructions to build a living organism. If we take this analogy to software code, the structures (e.g. LOC and coupling) within the code hold the DNA of the code. For the purposes of this dissertation, code DNA is the Java abstract syntax tree.

2.2.2 The AST and Applying DNA Sequencing to Software Code

An abstract syntax tree (AST) is a formal tree representation of the structure of the source code written in a programming language [Jones 2003]. The construction of ASTs normally involves the use of parsing

[10] Chromosome 21 is one of the 23 chromosomes that normally exist in pairs in the human genome.
[11] Electrophoresis is the motion of dispersed particles relative to a electrified fluid [Lyklema 2005]

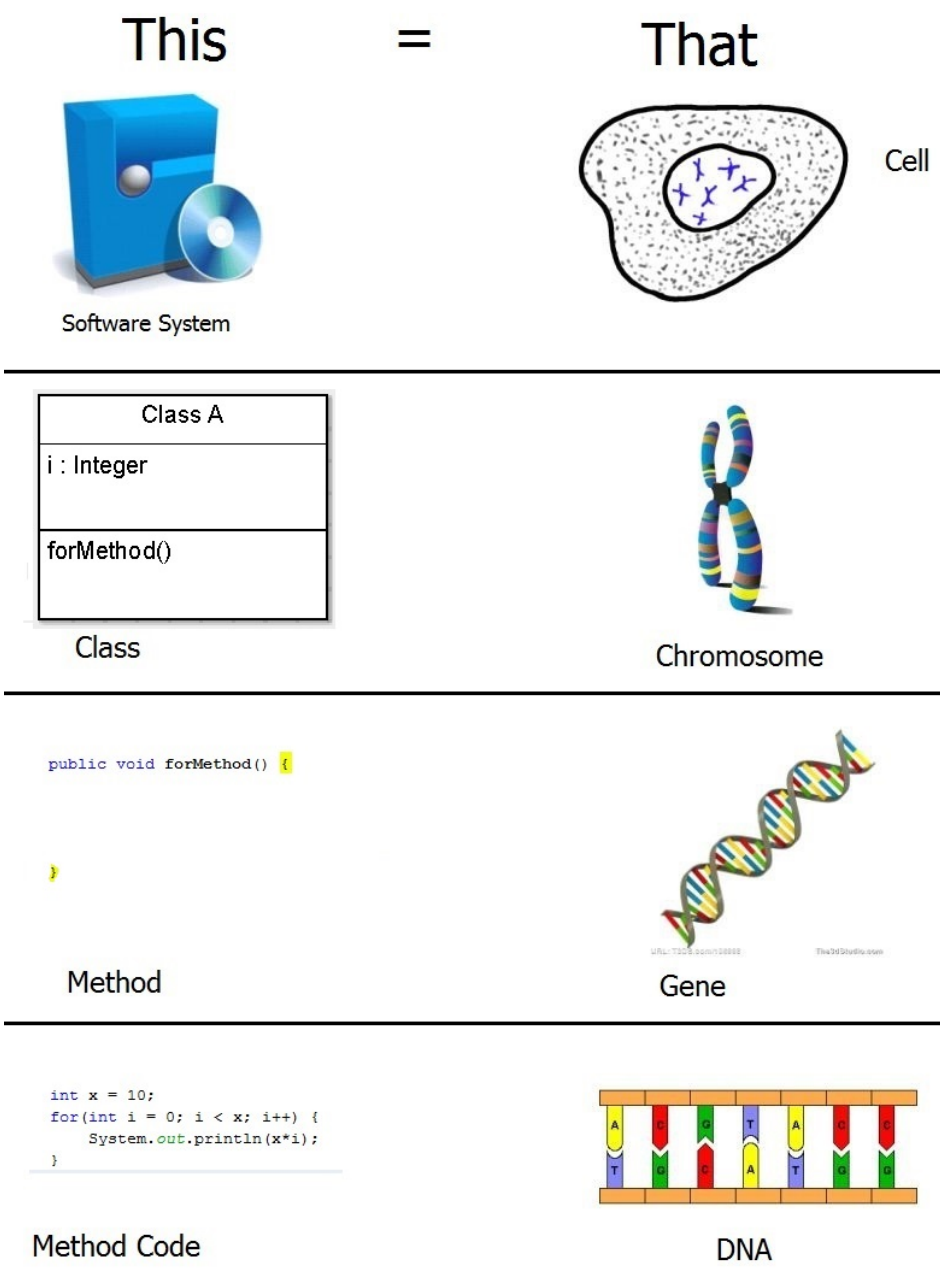


Figure 2.10: A graphic to show the software product to living organism analogy. In the first instance a software product/package is the cell in an living organism. These cells are made up of chromosomes, just like a software package is made up of classes. These chromosomes are made up of different genes, while a class is made up of different methods. Finally, the genes are sequences of DNA, like a method is made up of code.

technologies [IBM et al. 2013], with the tree consisting of nodes and branches. In computing the AST is used in a parser as an intermediate between a parse tree and a data structure [IBM et al. 2013]. ASTs are used to capture the structure of the source code and therefore omit unnecessary syntactic details (e.g. the identifier names and variable values) [Jones 2003]. This means an AST does not have the problems

associated with text as it does not hold information on the values or text representation of a construct. Meaning that human generated identifiers which could be specific to an individual text language, are not a problem with the AST.

This dissertation will use the Java Oracle AST in order to extract the programming constructs that have been used in the Java systems selected. The Java AST has been used as it is easy to identify a sequence of recurring constructs in Java source code (e.g methods). The nodes in the Java AST are known as kinds. In Java there are 92 different kind types[12]. Examples of the tree kinds are `For`, `If` and `Variable` (Appendix A.1 has the full list and their descriptions).

Figure 2.10 shows how a Java software system can be represented as a living organism's cell. The software system artifact that is used by the end user is like the cell within a living organism. This cell contains many chromosomes, like a software system is made up of classes. Each of the classes could be made up of a number of methods, while a chromosome is made up of many genes. The methods within a class will hold the instructions to which the program must adhere in order to run, whilst genes contain the DNA sequences and these DNA sequences hold the genetic instructions to build that living organism.

As the AST provides the entire formal structure of the code, this structure can be represented as a sequence of kinds. This kind sequence can be created to mimic a DNA sequence. Instead of a sequence of the four base pairs that are in DNA, the kind sequences will consist of combinations of the 92 available Java kinds. Each Java kind is a tree node of the AST and represents a different programming construct. The sequences of Java kinds are combinations of the names of each tree nodes that have been used to develop a system. Using the Java AST the code can be broken up into fragments, just like with DNA fingerprinting. These fragments can be the kind types that are present when you create the AST. In this dissertation the restriction point will be the `METHOD` kind. Southern blot graphs can be created by counting how many kinds are between certain restriction points. Figure 2.9 shows the amount of kinds for four different methods within EJDT 3.0, the greater the number of kinds within the method, the bigger the circle. How the methods are sequenced into kinds is described in more detail in Chapter 5.

2.2.3 Previously Found Patterns

The aim of the dissertation is to find Java code snippets which are associated with faulty code. This is a novel way of providing information on faulty patterns within software source code. Software patterns in the past have been used to warn programmers against bad coding practices. Fowler and Beck [1999] coined the phrase "software code smell" to indicate a pattern in the code that could relate to a deeper problem within a software system [Fowler 2006]. These smells are not indicators of problems on their own, but could point to an underlying problem within the system [Fowler 2006]. Fowler and Beck [1999] presented 22 code smells and suggested that they are a hint to decide where and when the software code should be refactored. Examples of code smells that could be represented as a code snippet are:

- Message chains - Message chains are when a object asks one object for another object. They are long sequences of method calls or temporary variables to get routine data (e.g. `a.getThis().getThat();` or in code snippet form - `METHOD_INVOCATION; MEMBER_SELECT; MEMBER_SELECT`). They are an example of coupling between classes.

[12] http://download.oracle.com/javase/6/docs/jdk/api/javac/tree/com/sun/source/tree/Tree.Kind.html

- Switch Statements - A switch statement is a structure that represents a structure control mechanism. This mechanism allows a variable to change the control flow of the program execution. In Java a switch statement will take a value and act essentially as a if/else statement. Which could look like `SWITCH; CASE; CASE;` as a code snippet.

Some of the 'bad smells' have been shown to be a problem in software engineering and are a cause of faults within a software system [Hall et al. 2014]. Shippey et al. [2012] investigated which papers had tried to associate code clones and defective code. Code clones have been shown to be a problem for developers as the inconsistent changes to them cause faults [Juergens et al. 2009]. Code clones however have also been shown not to be involved in defective data. Krinke [2008] showed that code clones are more stable than non cloned code as it is changed less often. Non cloned code also has a higher percentage of additions and removals to that of cloned code. Selim et al. [2010] used survival analysis using metrics to determine the impact of cloned code on software defects and which characteristics of cloned code have the highest impact on software defects. They discovered that cloned code is not more risky than non-cloned code. Göde and Koschke [2011] evaluate the evolution of mature software systems to determine inconsistent changes and risky code. They conclude that code clones are rarely changed and few have a high threat potential. Rahman et al. [2010] looked at whether cloned clone contributed to defects. Their findings suggest that most clones have little to do with defects and that cloned code contains less defective code than the rest of the system. The authors also discovered that large cloned groups have a smaller defect density per line than smaller clone groups. Zhang et al. [2011] investigated all the 22 code smells to see if there was evidence that they actually caused trouble within software systems. They found four papers that investigated the association between code smells and defects. Large classes, long methods have been significantly associated with software faults [Zhang 2009]. This has also been seen in defect prediction studies that have used LOC to show that the greater the LOC, the more chance a defect is to appear. Shotgun surgery was also significantly associated with software faults, but data classes, refused bequest and feature envy were all shown to not be significantly associated with software faults [Hall et al. 2014, Zhang 2009]. Code snippets could potentially be used to determine the location of the code smells introduced by Fowler and Beck [1999]. The significant association between a code snippet and a fault could provide additional evidence to the association of bad smells and defects. Finally, there could be additional bad smells that could be discovered by examining the associations between code snippets and faults. The code snippet technique could have a potential advantage over the code smell approach. The code snippet technique does not rely on human conjecture and will highlight an unbiased list of all the possible coding constructs that have the potential to introduce defects into a software system.

2.3 Summary

This chapter has provided background information on the research that is to be undertaken in this dissertation. The chapter has defined what a defect is and the costs involved with defects within software engineering. The chapter has described what software defect prediction is and has outlined how it is undertaken. Defect prediction relies on both dependent and independent variables. This chapter has defined what an dependent variable is and how dependent variables could be extracted from a software system. The traditional independent variables that have been used in defect prediction have been introduced. These independent variables rely on information both from the source code and the processes around storing the source code (e.g. the version control systems). This chapter has highlighted the

problem with traditional independent variables. The traditional independent variables have been shown to have reached a ceiling on their predictive power and new variables may be needed. These new variables could be based on finer grained independent variables. This chapter has discussed the current finer grained independent variables available within defect prediction. Most of the new finer grained independent variables are based on the source code text, however, some use subgraphs to describe the code. The finer grained methods have shown promise in defect prediction. One of the better finer grained methods uses a subset of code features based on operators. The approach that is to be introduced in this dissertation goes further and analyses all the available low level code features within the source code. This method is based on the AST and is motivated by work within biology. The chapter describes how the work within DNA and genetic disorders has inspired the work introduced in this dissertation. The chapter describes that DNA is the building blocks of an organism and how new finer grained genetics tests based on the differences in DNA can identify genetic disorders easier and faster, than the more coarse and older methods. This chapter has described the relationship between this DNA analysis and how it is related to code. It has described what an abstract syntax tree is and how it will be used to extract sequences from source code. Finally, the chapter has highlighted some of the current patterns within code that have been found to be related to defects. These patterns are known as code smells, and there is evidence that these code smells have a negative impact on the stability of the code.

Chapter 3

Methodology

This chapter will introduce the methodology that has been used to answer the research questions in this dissertation. Section 3.1 describes the research methodology chosen, why it was chosen and the pros and cons of this methodology. Section 3.2 describes the research methodology protocol. Finally, Section 3.3 introduces the systems which have provided the data on which the methodology is applied.

3.1 Research Methodology

The main research methodology used in this dissertation is an empirical research methodology. Empirical research methods *"are a class of research methods in which empirical observations or data are collected in order to answer particular research questions"* [Moody 2014]. Empirical research normally follows five different stages. These five stages were introduced by de Groot and Spiekerman [1969]. Figure 3.1 shows the cycle of these five stages. The five stages of the empirical life cycle as described by de Groot and Spiekerman [1969] are:

1. **Observation** - The collecting and organisation of empirical facts to use in forming an hypothesis. The observation phase is where a researcher will build an idea that they will later test and conclude on. During the observation phase the researcher is relatively free on what research they are doing or what they intend to do. For example, if a researcher was carrying out an extension to a previous study, during the observation phase, the researcher will look at the previous study, and observe ways to extend the study. There could be various ways the researcher could extend the study, for example, choosing different datasets, a different technique or a new programming language. The observation phase will help the researcher decide which route to follow.

2. **Induction** - The formation of the hypothesis based on the previous observations. Based on what the researcher has viewed in the observation phase, they have to devise a question that is to be answered in the study they are about to carry out. The researcher has to decide if the hypothesis is testable and if it is not can an answer be drawn from the research.

3. **Deduction** - Formulate the hypothesis to create testable predictions. To formulate the hypothesis the researcher must present a hypothesis that is logically consistent within the research framework, it must be achievable within the budget, the hypothesis must be testable and finally the scope of the theory must be well defined.

4. **Testing** - Testing the hypothesis with the new empirical material. This is the results section of a researcher's study. The testing is where the researcher would provide the evidence to either reject their hypothesis or accept there was no evidence to do so.

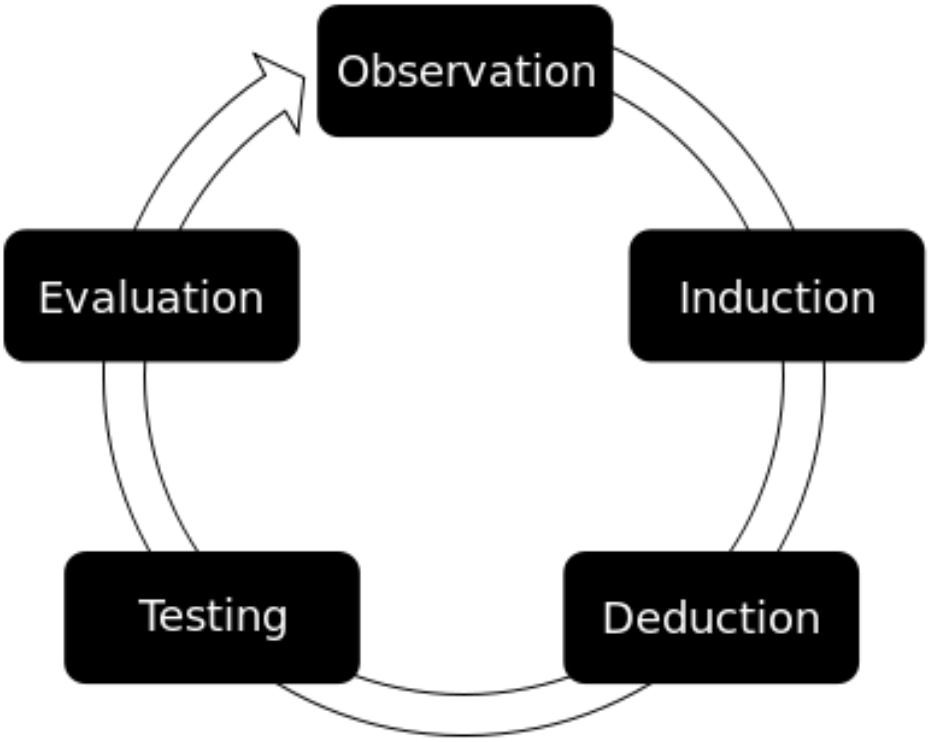


Figure 3.1: de Groot and Spiekerman [1969]'s empirical methodological cycle. Taken from [Daan 2014]

5. **Evaluation** - Evaluating the outcome of the testing. The evaluation stage is where the research will conclude the findings of their research. The evaluation phase will provide what has been learnt during the research and what the results mean to the wider research community. Also in this phase the researcher has to decide how the results are to be published. This is the final stage of the research, however the conclusions of this researcher's research could lead to the observations of another researcher's research.

Empirical research methods can be divided into two categories; qualitative or quantitative research. Qualitative research methods collect qualitative data. Qualitative data is a categorical measurement expressed by a language description, not numbers. Qualitative research normally aims to gather understanding of human behaviour and the reasons that govern that behaviour [Denzin and Lincoln 2011]. Examples of qualitative data include but are not limited to text, images and sounds from observations, interviews or documentary evidence. In software engineering examples of qualitative methods that are primarily qualitative are ethnography, case studies and action research [Easterbrook et al. 2008]. Ethnography is where a researcher will study a group in a natural setting over a period and collect observational data [Creswell 2003]. Case studies are when a phenomenon is investigated in its real world context [Easterbrook et al. 2008]. Case studies offer an understanding as to why and how certain phenomenon occur and the cause and effect relationship [Easterbrook et al. 2008]. A variety of data is collected during case study research, including interviews and observations. [Kitchenham et al. 1995] offers guidelines on how to conduct case studies. According to Easterbrook et al. [2008] there are two types of case study

- exploratory and confirmatory. An exploratory case study is an initial study, that will look to discover new hypothesis and build new theories. A confirmatory case study is used to test existing theories. Action research is where researchers attempt to solve a real world problem, whilst studying the experience of solving the problem [Easterbrook et al. 2008]. Action research attempts to provide practical value to a client organisation whilst simultaneously contributing to the gaining of new knowledge [Sjoberg et al. 2007]. Grounded theory is where the researcher tries to derive a general theory of a process or interaction based on the views of the participants in the study [Creswell 2003]. Grounded theory involves different stages of data collection in order to reconfigure categories of information. The grounded theory process involves constant analysis of the data versus the categories and sampling of different groups in order to maximise the similarities and differences in the information Creswell [2003]. Some of the advantages of qualitative research are that it is flexible and can easily evolve throughout the study, it looks at a situation in more detail than just looking at the numbers, it can provide new topic areas as the questions posed will be encouraged to expand on their responses and qualitative research attempts to avoid creating pre-judgements of a situation. The negatives of qualitative research are that usually the sample sizes are very small and the information gained cannot be generalised to a wider population, the collection of quantitative data is usually very time consuming and therefore could make the research more expensive and it could be harder to make comparisons as the subjects may give a wide range of different responses.

Quantitative research methods collect numerical data and analyse it using statistical methods. Quantitative research focuses on developing hypotheses to explain the things that happen, and uses mathematical models and statistics to help answer these hypotheses [Given 2008]. Examples of quantitative research include experiments, where the researcher could apply a new treatment to a patient and then observe the results to see if there is a causal relationship between the treatment and recovery rate. A survey is another example quantitative research method. In a survey, questions are asked to a panel of people via interviews or questionnaires and the aim is to answer a hypothesis depending on their answers. Another example is to look at historical data and observe what has happened and see if patterns emerge. These patterns could be used in the future to direct decisions (e.g. Investment patterns). Quantitative research normally follows the same pattern. A researcher will create an hypothesis to answer a specific question, sample numerical data is collected in order to answer this question and then this data is analysed using statistical techniques. The aim is that the answer gained from the statistical techniques on the sample data can be used to generalise to the wider population [Given 2008]. Quantitative research allows for the involvement of a greater number of subjects, which can enhance the generalisation of the findings. Other advantages of quantitative research include - the use of standards mean that the research could be replicated or compared with similar studies, and personal bias can be avoided as less time is spent with subjects.

There are examples when qualitative and quantitative research methods have been combined in the same research procedure, known as a mixed method approach. For example the use of observations and interviews are combined with traditional surveys [Creswell 2003]. Mixed methods were introduced as researchers recognised that all methods have their own strengths and weaknesses. The use of both quantitative and qualitative methods in one study could reduce the amount of bias that could occur if only one method was used [Creswell 2003]. There are three main mixed method strategies - sequential, concurrent and transformative procedures [Creswell 2003]. Sequential procedures are those in which a researcher will expand on the findings of one method with the other. In a concurrent procedure a researcher will collect both qualitative and quantitative data at the same time to comprehensively analyse the research problem. A transformative approach could include either a sequential or concurrent approach.

3.2 An Empirical Study Exploiting Abstract Syntax Trees to Locate Software Defects

This dissertation is an empirical study investigating if a significant relationship exists between Java construct patterns and defectiveness, and how this relationship changes over time. The dissertation methodology will be broken down into the three of the five stages introduced by de Groot and Spiekerman [1969]. The evaluation phase is covered in Chapters 8 and 9, the discussion and conclusion.

3.2.1 Observation and Induction

Chapter 2 has highlighted the need for a metric that can break the current performance ceiling. This observation of background research has shown the different ways in which researchers have tried to find associations between independent variables and defects. The observation phase has also included the investigation of how defects are found within biology and that changes in an organism's DNA can lead to defects. The observation of the current ceiling in defect prediction and the way in which defects are currently predicted in human beings has led to the idea of code snippets within a software system. Software code is to be broken down in the same way a living organism is, into small constructs, known as code snippets.

3.2.2 Deduction - developing the hypothesis

This dissertation will try and establish an association between code snippets and faults. The use of software code to create code snippets is a new technique within defect prediction and as such, the experimental method chosen is important. The deduction phase saw the formulation of the research questions seen below, these research questions were devised to answer specific questions in regard to the association between code snippets and faulty code. The research questions are answered by way of a statistical test.

RQ1 Are any code snippets significantly associated with;

(a) faulty code?

(b) non-faulty code?

RQ2 Do the associations between code snippets and faultiness change as a system evolves?

These research questions will be answered by the collection and analysis of data. This data is various selected systems source code and defect data. The independent variables needed will be processed from Java source code and the dependent variables processed from defect data. The source code data collected will be processed in order to create the independent variable, the code snippets. These code snippets are needed to test whether there are associations between specific Java constructs and the dependent variable, the faults. The association between defects and the code snippets will be investigated by statistically testing if the appearance of a code snippet within a set of defective methods is not likely to be by chance. Section 3.3 outlines the criteria for selecting a system to be chosen for use within the dissertation and which systems were chosen.

3.2.3 Testing - Finding the Sequences and Significant Associations

The testing phase is where the evidence is provided to answer the research questions generated in the previous phases. To provide this evidence this dissertation introduces an approach that could identify possible Java constructs that are associated with faults. Figure 3.2 gives an overview of the processes that were undertaken in this dissertation. The method has two main strands; one is the identification of the independent variables - the code snippets, and the other is discovering the dependent variables - the fault data. Identifying the independent variables is the strand on the left on Figure 3.2. To identify these independent variables, Java source code was turned into method sequences using the modified pretty printer. Code snippets were extracted from these method sequences. The dependent variables were identified using the strand on the right hand side of Figure 3.2. To identify the dependent variables, a defect repository and a Git repository were needed to create the defect fix links and these defect links are used to find the defect insertion points (Section 4 has the full detail on how defect fix links and defect insertion points are discovered). This information will allow for the labelling of methods being defective or not at a particular release of a system. These labelled methods and their corresponding code snippets are then used to determine association with faults using the binomial test. This test will produce two sets of code snippets, those that are significantly associated with faults and those that are not. Chapter 6 has the full details of how this is carried out.

3.2.3.1 Identifying the Independent Variables - The Code Snippets

One of the main processes is to extract the independent variables from the source code of a chosen software system. The independent variables to be extracted are the Java code snippets. The Java code snippets are small sequences of Java constructs based on the Java AST. To extract these code snippets, the source code has to be broken up into modules which are based on a particular restriction point. For this dissertation, the restriction point is at method level. This means that each of the methods found in a particular system will be sequenced into an ordered sequence of Java kinds.

To build a corpus of Java method sequences for each of the five systems investigated, the entire project was compiled using the Oracle JDK parser[1]. When a class is compiled the JDK creates an AST. This AST is traversed via the use of an extended `Pretty.java`[2] class. This extension helps store Java kinds[3] in the order they appear in the AST. This is completed for each of the methods found in the AST and creates a corpus of sequenced methods for each class and ultimately each system. The Oracle AST has been used as it is well documented and it is easy to turn a source code file into a traversable AST.

The use of the AST or a lexer has been used extensively in code clone detection techniques. A code clone is a where two pieces of code in a software system are identical or similar. A lexer breaks down the text of the source code into a sequence of tokens [Roy and Cordy 2007] and then these tokens of two pieces of code are compared to find the clones. This technique was used to make two prominent code cloning tools - CCFinder [Kamiya et al. 2002] and CP-Miner [Roy et al. 2009]. The AST is used for clone detection by processing the trees with either tree-matching methods or structural metrics [Roy et al. 2009]. The AST has been used in many techniques [Baxter et al. 1998, Raza et al. 2006, Wahler et al. 2004] which have reported good precision and recall [Roy and Cordy 2007] showing that this approach is effective at differentiating features of code.

[1] http://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.html
[2] Pretty.java is found in tools.jar of the Oracle JDK and uses the Visitor pattern to walk an AST
[3] A full list of the 92 Java kinds and their meanings is found at in Appendix A.1

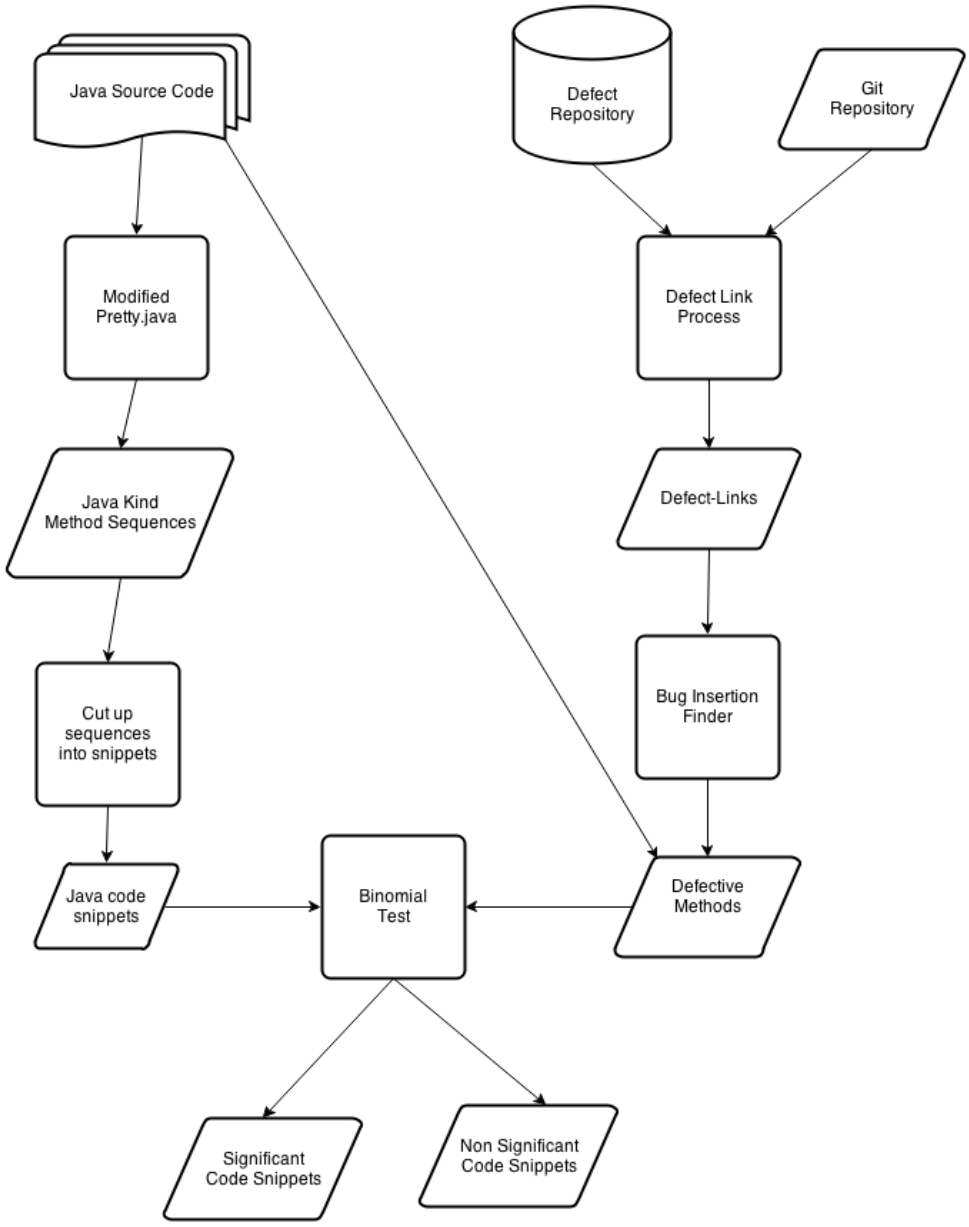


Figure 3.2: Overview of the process that will be undertaken within the PhD.

The AST has been used in this dissertation to break the code into different kinds, as it provides a good number of different constructs and these constructs are easily understandable. Breaking down the code using an AST, instead of manually creating a lexer, or using the text of the source code was undertaken because of a number of issues. One is that source code text is not always consistent, and creating rules for the differing number of possible ways that a construct could be created or the different languages that could be used, is not efficient. The AST already breaks down the code into constructs, there is not a need to try and 'reinvent the wheel'. Another is that by using a publicly available parser, another researcher could easily recreate the method that is presented in this dissertation, without the need to create or borrow a custom made lexer. This helps the research presented in this paper be more reproducible. Which is an important factor in empirical research studies as suggested by Kitchenham et al. [2002].

The sequenced methods are used to find all the different Java *'code snippets'*. To find the code snippets each method is broken down into subsequences of kinds of differing lengths. From one kind to a maximum of seven. This maximum could be changed to any length required. For the purposes of this dissertation, a maximum snippet length of seven was used. A maximum of seven was chosen due to computational restraints. These code snippets are stored in a MySQL database and labelled with which method they belong in.

3.2.3.2 Identifying the Dependent Variables - The Faults

The dependent variables involved in this dissertation are the faulty methods of the specific software systems. This dissertation is only interested in whether a method is faulty or not at a particular system release. The number of defects in that method is beyond the scope of this research and can be covered by further work. This means that the dependent variable is a binary value; defective or not defective. This has been chosen as it is the association of faults and code snippets that is being investigated, not the number of defects a particular method may or may not have. To create a database of faulty methods, it is necessary to find where the faulty methods are in each of the systems analysed. Faulty methods are found using their fault-insertion and fault-fix points.

To identify faulty methods, there are two different techniques that could be used; *'SZZ'* by Śliwerski et al. [2005] (Chapter 4 has the full explanation of the technique) or *'ReLink'* by Wu et al. [2011]. SZZ has been used in this dissertation above ReLink due to ReLink taking a long time to complete on the systems studied. SZZ is a popular algorithm for its purpose and has been used in many different studies [Fischer et al. 2003a, Kim et al. 2006; 2007, Čubranić and Murphy 2003, Williams and Spacco 2008, Zimmermann et al. 2007]. The implementation of SZZ used in this dissertation was verified by manually checking ALL bug links found for a particular system release. The implementation has been shown to have good recall and near perfect precision, Chapter 4.2 presents the results of this analysis.

SZZ is an algorithm which matches the fault fix described in the bug tracking system with the corresponding commit in the version control system that 'removed' the fault. By backtracking through the version control records, it is possible to identify earlier code changes which ended up being 'fixed'. It is assumed that the earlier code change inserted the fault. The method is therefore labeled as faulty between the time the fault was inserted and the time it was fixed. Using this technique it is possible to identify for a particular snapshot of the code base, which modules are faulty and which are not. There will be defective modules which have not yet been reported. It is therefore important to carry out the fault mapping after sufficient time has passed for users to report faults. It is unlikely that all defects will be reported and therefore there will be false negatives. Kim et al. [2011] suggests that as long as the number of false negatives and false positives is less than 20% in total, the results for defect prediction that are determined from that dataset are reliable.

3.2.3.3 Determining if Associations Exist between Code Snippets and Faults

The independent and dependent variables that have been gathered by the previous methods are used to answer the main research questions that this dissertation answers. The code snippets are statistically analysed to determine if any code snippet is associated with faulty or non faulty code. To find if a snippet was associated with either faulty or non faulty methods the percentage of faulty methods is calculated. The percentage of faulty methods was calculated for the entire system and then for subsets of methods

	Criteria	Reasoning
1	The number of trials must be fixed	A trial is whether a method is defective or not. The number of trials is the number of methods that the code snippet appears in. This is fixed as a code snippet will appear in a finite number of methods within each system.
2	Each trial must have the same two possible outcomes	The two possible outcomes of a method trial is whether it is defective or not. Each method in a system will have one of the two outcomes.
3	The trials must be independent	The outcome of a method being defective is not reliant on another being defective.
4	The probability of success must be the same in each trial	The probability of a method being defective is the same for each method. It is the overall defectiveness of the system. This is set at the start of the test.

Table 3.1: The criteria for the binomial test [Garson 2004] and how the data used in this dissertation passes the criteria.

which contained each snippet. The percentages of faulty methods are compared using the binomial statistical test to determine which snippets are significantly associated with either faulty or non faulty methods. The binomial test compares the observed count for each snippet to the count that would be expected under the assumption of no association between the snippet and defectiveness. Equation 6.1 in Chapter 6 has an example of how the binomial test was used in this dissertation.

The binomial statistical test is used as the data adheres to the rules of the binomial test. There are four criteria that must be adhered to whilst undertaking the binomial test. Table 3.1 shows the criteria and the reasons for the data used in this dissertation passing the criteria. It is important to state that a test is performed for each snippet. If the trials (method is defective or not) chosen were at random, it would be expected that the proportion of the sample would not be significantly different to that of the overall population. If the trials that have a particular code snippet within them are shown to be significantly different to the overall population, there is evidence that there could be an association between the faults and the particular snippet.

This approach relies on hypothesis testing. Hypothesis testing is the use of statistics to determine the probability that a given assumption is true. In statistics it is used to test a statistical assumption. The tests are used to determine if a test result is statistically significant or it has occurred due to sampling error/by random chance, using a significance level. Usually, there are two types of hypotheses - a null and alternate hypothesis. A null hypothesis (H_0) is one that a sample has occurred purely by chance. An alternate hypothesis (H_1) is the hypothesis that the sample observations are influence by some non-random cause. Hypothesis testing consists of four steps:

1. State the hypotheses - outline the null and alternate hypotheses. These must be mutually exclusive.
2. Analysis plan - The analysis plan describes how to use the sample data to evaluate the null hypothesis. This normally revolves around a test statistic and a decision rule. The test statistic is the

single measure of some attribute of the sample. Examples of a test statistic are the mean, z-score, proportion or t-score (there are many more than can be used). The decision rule can be either a p-value or an region of acceptance. The p-value is the probability of observing the test statistic. A region of acceptance is a value range which if the test statistic falls within this range, there is no evidence to reject the null hypothesis. If the statistical test value falls out of this region, then the null hypothesis is rejected.

3. Analyse the sample data - This step involves calculating the value of the test statistic chosen in the analysis plan.

4. Interpret the results - Apply the decision rule described from the hypothesis tests. The decision rule normally involves If the test statistic is lower than the significance level, there is evidence one can reject the null hypothesis. This means that the sample data could be influenced by a non random cause. If the test statistic is higher than the significance level, there is no evidence to reject the null hypothesis and the sample data could have just happened by chance.

This dissertation has followed the four steps outlined above. The null hypothesis for our data is that the code snippets are not located in a higher amount of defective methods compared to the population. The alternate hypothesis is that they are in a higher proportion of defective methods than what is expected and therefore they are significantly associated with faults. Chapter 6. The test statistic is the binomial test score and the significance level was set to 99%. The four steps undertaken in this dissertation are described in more detail in Chapter 6.

3.2.3.4 Tools Created during the Testing Phase

For two of the three phases of the testing phase (discovering the faulty methods, identifying the code snippets) a tool had to be created from scratch to automate each phase.

To discover the faulty methods, an implementation of the SZZ algorithm had to be created. This tool was developed from scratch using the information provided in the Śliwerski et al. [2005] paper. The original Śliwerski et al. [2005] was modified due to problems that were encountered in the testing phase. This is discussed more in Chapter 4. It had to be created from scratch because no current implementation could be found on the internet. The tool was created in the Java programming language and can download defect reports from a defect repository and can connect to a Git or SVN repository. The tool can determine the defect fix and possible defect insertion points for any program with a defect database and Git or SVN repository. The tool also allows for CSV input.

The method sequences and their respective code snippets were extracted from the source code using another tool which I developed in Java and from scratch. There is no current tool available that allows for the sequencing of Java code and extraction of code snippets. The tool can take a source code repository and turn each of the methods into a method sequence. The tool then extracts all the code snippet into a MySQL database.

3.2.4 Problems in Empirical Research

As with most research methodologies, empirical and quantitative research has its limitations. This subsection will describe these limitations, but will also provide solutions to how these limitations were

mitigated, if possible, during the dissertation. The issues affecting good empirical research can be broken down into the following categories; the data used, the experimental design, experimental context, external forces and time and money constraints.

Kitchenham et al. [2002] introduced preliminary guidelines for conducting empirical research within software engineering. Kitchenham et al. [2002] introduced these guidelines as in their "view, the standard of empirical software engineering research is poor" [Kitchenham et al. 2002]. This view is not limited to the software engineering world, with many other studies finding poor empirical research being conducted [McGuigan 1995, Welch II and Gabbe 1996, Yancey 1990]. Kitchenham et al. [2002] believed the guidelines were important as software researchers often made statistical mistakes and that senior researchers were pressing for more empirical research to underpin software engineering. The guidelines can help researchers improve their research planning, implementation and reporting [Kitchenham et al. 2002].

The data used in a research study is of little use without proper analysis. Just presenting the data or applying curve fitting algorithms is not enough. It is possible to summarise the data and not report all of the data, Perry et al. [2000] states that a researcher could just provide the upper and lower bounds and that the precision of the data should relate to the problem context. The data is the basis on which to answer the main research questions, not fill space in a study [Perry et al. 2000].

One problem with collecting data for a research study is the time and cost that it could undertake to collate the data needed [Gagnon 1982]. For example, a researcher could have to go out into the field to collect the data at software firms. The costs involved in collecting this data could include travel and possible accommodation to and from the firms. The access to this data could also be limited as businesses could not allow all the necessary data that the researcher needs in order to answer their hypothesis, or access to the data could be denied completely by the firms involved [Gagnon 1982]. There are no costs involved with collecting the data in this dissertation as they are all available via open source repositories.

The experimental design of the research needs to be clear, concise and reproducible. Perry et al. [2000] states that the most important part of the research is to draw a conclusion from the results of the research. These results need to be causal, actionable and general. A causal relationship needs to be explained clearly by the results [Perry et al. 2000] (e.g. an increase in sales, leads to an increase in profits). The results must be actionable as the researcher could then carry out further work to extend the research, or the research could be put into action [Perry et al. 2000] (e.g. increase sales in a product to increase profits). Finally the results must be general, this means that the study should provide conclusions that could be used in a wide variety of contexts [Perry et al. 2000] (e.g. an increase in sales in the automotive industry that leads to increased profits, could be applied to the technology sector). The experimental design needs to be clear and concise, this is so the experiments carried out by the original researcher can be easily replicated by another researcher [Kitchenham et al. 2002, Perry et al. 2000]. This means that data and research procedures should be made public. The methodology used in this dissertation is as clear and concise as possible. This dissertation provides all the possible information required to replicate the research. The data chosen is open source and the chosen AST to create the sequences is the default Oracle parser. Both of which are available to all other researchers. The tool that is used to create the bug-links between the defect reports and the commit messages will be made public for other researchers to use.

One problem that is raised by both Kitchenham et al. [2002] and Perry et al. [2000] is that the use of proper statistical tests needs to be taken seriously. Kitchenham et al. [2002] highlights that a lot of researchers do not understand the statistical tests properly and as such are applying them in the wrong

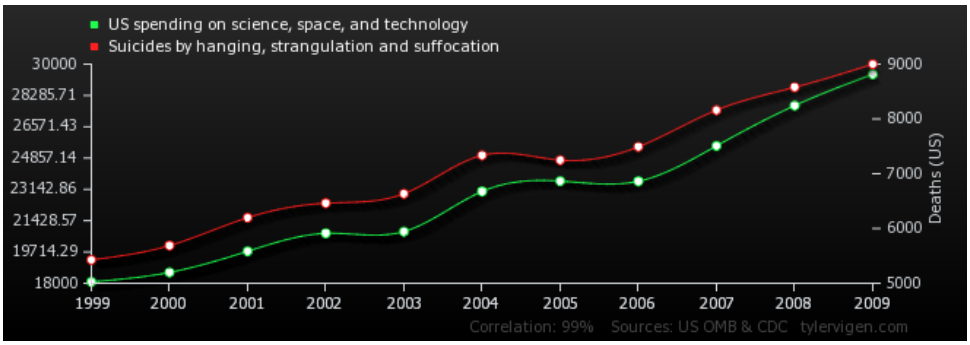


Figure 3.3: The correlation between the spending of science, space and technology in the US and the suicide by hanging, strangulation and suffocation rate in the US is 0.99. This is very high, but obviously the two are not causally linked. Taken from SpuriousCorrelations [2014]

contexts. The problem is exacerbated by the fact that reviewers of these studies are also unsure of the correct statistical tests that should be used and as such, cannot correct the original research of their mistakes [Kitchenham et al. 2002]. The use of the wrong statistical methods being used can lead to incorrect results being reported. One example of a researcher not understanding a statistical technique correctly is the *"correlation does not imply causation"* problem [Aldrich 1995]. This is where a correlation between two sets of data does not necessarily mean that the two are linked[4]. Figure 3.3 taken from SpuriousCorrelations [2014] shows one example of these spurious correlations; the spending of science, space and technology in the US, which correlates highly with the suicide by hanging, strangulation and suffocation rate in the US. Which would mean that if the US government increases its spending in science, space and technology that deaths by suicide should increase. Obviously, despite the high correlation, one does not cause the other. It is important that the researcher fully understands the data they are working with and that they can explain the results they are extracting. This leads to the studies being undertaken needing to have high validity. The studies need to establish credible solutions or answers to problems (a decrease in NASA spending will probably not lead to a decrease in suicides!). The statistical method carried out in this dissertation has been carefully selected. The reasons for choosing the binomial statistical test are defined in the previous section. The results will also be discussed in detail, so the causality of the results can be explained. Experience in using the datasets and with previous results regarding some structural patterns and code will possibly help explain some of the results.

When designing an empirical study the real challenge is to design a high-impact credible study [Perry et al. 2000]. Perry et al. [2000] states that this means there must be trade-offs so that the researcher maximises: the accuracy of the results, so that the results are not a factor of unknown influences; that the results tell the researcher and the wider community something important about the subject area; finally, the results of the study affect further studies or encourage further research in the subject area.

3.3 Systems Providing the Data for this Dissertation

To analyse the association between particular code snippets and defects this dissertation required the use of source code and defect report repositories. To gather this information, different systems needed to be chosen. For a system to be selected for use in this dissertation it had to adhere to certain criteria. The

[4]There is a website dedicated to proving this correct - http://www.tylervigen.com/

system:

1. **has to be developed in Java** - The system has to be predominately coded in the Java programming language. Java was chosen as it is easy to extract an AST and it is a language that I am proficient in.

2. **has to be mature** - The system had to have many releases and revisions. The system has to be mature as it needs to have many faults recorded and many different changes to the source code in order to gather sufficient defect data. If a system is immature, then defects may have not been reported or may not have had a chance to be fixed.

3. **must be open source** - The system has to be open source as access to the source code and bug repository is required. Open source code is easier to attain than closed source data and requires little costs.

4. **have a Git source code repository** - The system must be under management by a Git source code repository. Firstly the system must have a source code repository as the SZZ technique requires the mining of the history of the source code. Secondly the source code repository must be a Git repository as the SZZ implementation has been created to mine a Git repository. Git does not require a central repository like SVN and CVS. This requirement does not have to feature in future research. In theory any software repository could be used.

5. **have a defect database** - A defect database is required as the SZZ technique requires access to the fault reports that have been created when the system fails.

Using the above criteria, five Java systems were selected for use within this dissertation. In addition to the criteria these systems were chosen as they had previously been extensively used in previous defect prediction studies [Hall et al. 2011, Shippey et al. 2012]. The Java open source programs used to study the association between faults and code snippets are; Eclipse.JDT.core (EJDT), AspectJ and ArgoUML. Table 3.2 shows how each system met the criteria set out above. To study RQ2 three different versions of EJDT were examined. Table 3.3 shows the contextual information for each of the systems studied. Each of the systems is described in more detail below.

System	Age (Years)	Releases	Revisions	Repository	Defect Database	Open source license
EJDT	13	47	19,357	Git	Bugzilla	Eclipse Public License 1.0
AspectJ	12	37	7,609	Git	Bugzilla	Eclipse Public License 1.0
ArgoUML	14	19	17,749	Git	Issue Tracker	Eclipse Public License 1.0

Table 3.2: The criteria assessment for the three programs analysed in this dissertation. Each program is a mature open source program with a Git repository and a bug database.

System	Release	KLOC	Avg Line Length	Total Methods
EJDT	2.0	216	33	10,927
EJDT	3.0	292	34	13,885
EJDT	3.1	349	35	16,052
AspectJ	1.7.0	353	33	21,980
ArgoUML	0.20	273	34	12,330

Table 3.3: The five systems analysed in this dissertation.

3.3.1 Eclipse.JDT.core (EJDT)

Within software engineering, Eclipse is an "open source community of tools, projects and collaborate working groups" [TheEclipseFoundation 2014b]. One of the most commonly known tools that Eclipse provides is an integrated development environment (IDE) for Java, C/C++ and PHP. The IDE is modular and allows the use of plugins to be used to extend the original IDE base. The IDE is made up of different projects, JDT is one of these projects.

Eclipse Java development tools (Eclipse.JDT) is a set of IDE plugins that allow the Java capability on the platform [TheEclipseFoundation 2014c]. The package Eclipse.JDT.core (EJDT) is one of these plugins. EJDT defines the non-UI elements of the plugin [TheEclipseFoundation 2014c]. EJDT includes a Java builder, an API that allows navigation of the Java element tree, and code assist[5] and a source code formatter. The EJDT package was chosen because the faults had previously been mapped between the bug tracking system and the version control system [Bird et al. 2009a, Śliwerski et al. 2005, Zimmermann et al. 2007]. This allowed the comparison of generated bug links, which is discussed in more detail in Chapter 4.2.

3.3.2 AspectJ

AspectJ is an aspect-oriented programming[6] extension to the Java programming language [TheEclipseFoundation 2014a]. AspectJ can be used to improve the modularity of a software system. It allows a user to create clean modularisation of crosscutting concerns such as error checking and handling, performance optimisations and debugging support [TheEclipseFoundation 2014a].

3.3.3 ArgoUML

ArgoUML an open source unified modelling language (UML) modelling tool [Collabnet 2014]. ArgoUML is written in Java code and has been downloaded over 80,000 times from all over the world [Collabnet 2014]. ArgoUML allows the design of UML diagrams to support the development and documentation of object oriented systems (e.g. use case, class and sequence diagrams).

[5]Code assist helps predict potential code that the user is trying to write as they are typing it. This automatically offers suggestions to complete the current code.

[6]Aspect-oriented programming is a programming concept that allows the separation of crosscutting concerns to increase modularity.

3.4 Summary

This chapter has provided an introduction into the research methodology used within this dissertation. It has described what empirical and quantitative research is and the drawbacks of these research that have been identified. The chapter also outlined what important aspects should be taken into account when designing an empirical research methodology. The chapter has described the methods that will be undertaken in this dissertation to answer the research questions and why these methods have been chosen. It has also introduced the systems that have been used and outlined what these systems are and the criteria used to choose them.

Chapter 4

Code Repository and Defect Database Mining

In order to be able to predict defective modules, the defects in a software system have to be identified. This chapter outlines the process of finding the defective code used in this dissertation. The process uses repository mining techniques to identify links between a software repository and a defect database, a *bug-link*. A bug-link is an important part of finding defective code. This chapter will describe what a bug-link is, how they are extracted, how they can be used to find defect insertion points and how the insertion point is used to find defective code. This chapter will also highlight the difficulty in performing replication studies for data mining repositories and bug databases.

4.1 Mining Software Repositories

Herzig and Zeller [2010] defines mining software repositories as a process to obtain initial evidence by extracting data from software repositories. Software repositories are sources of information such as software code, defect databases and version archives. Mining software repositories is an active research area within software engineering and has its own conference: 'Mining Software Repositories (MSR)'[1]. MSR describes that the mining software repositories field "analyses the rich data available in software repositories to uncover interesting and actionable information" [MSR 2014]. MSR will have its 12th annual working conference in 2015. The data extracted by mining software repositories can be used to identify defective code within a software system.

One way to identify defective software code is to analyse the code's respective software version control system (VCS) and defect tracking system. A VCS records changes to different files when a developer has completed some work on the system's code. Each recorded change is called a commit and is identified by a number within the VCS. The commit is normally performed by the person performing the change and will have associated with it - time, the author and a message authored by the committer. Four well known examples of VCSs are CVS, SVN, Git and Mercurial. In this dissertation the corresponding Git repositories were used for each dataset. This is because of previous expertise gained when using Git.

When a problem is found within a program, it is good practice to put details of the problem into a defect tracking system. This entry is called a defect (or bug) report. The defect report holds the program or program part where the defect was found and sometimes a message of how it occurred. The defect is given a unique identifier within the tracking system. Examples of defect tracking systems are Bugzilla, Issue Tracker and Jira. A defect tracking system is needed because sometimes VCS data does not always include the information of why a change was made [Śliwerski et al. 2005]. When the defect tracking information and the VCS information is combined, the reliability of discovering bug links is increased.

[1] http://2014.msrconf.org/

If the defect tracking system was used alone, the commit that caused the change would not be known, only the time that the defect report was updated. Similarity, if just the VCS was used, the defect fix numbers that have been reported in a commit message may not actually be defect numbers.

One method to identify defective code is to match a particular commit where the defect has been fixed from the VCS to the correct defect number in a defect tracking system. This will give a bug-link. This bug-link will tell us where a particular defect has been fixed and is used subsequently to find the defect insertion point by tracking the historical changes to the fixed code.

The defect insertion point is an important piece of information on the lifeline of a defect. If the insertion point and the fix point (the bug-link) is identified for known defects, then it is possible to know where there is a defect at any point in the history of the code. Figure 2.4 shows how the life of a defect can be exploited to determine the insertion point.

To complete the above tasks, the algorithm first described by Śliwerski et al. [2005] was used as a starting point for developing a more reliable bug-linking technique. The technique was applied to five systems to find potential bug-links and their defect insertion points.

4.1.1 SZZ

SZZ is a bug linking algorithm described by Śliwerski, Zimmermann, and Zeller [2005]. SZZ was based on work by Čubranić and Murphy [2003] and Fischer et al. [2003a;b], who inferred links from references by using Bugzilla defect reports combined with CVS commit messages. SZZ is a frequently used algorithm and has been used in many studies [Fischer et al. 2003a, Kim et al. 2006; 2007, Čubranić and Murphy 2003, Williams and Spacco 2008, Zimmermann et al. 2007]. This algorithm is described in more detail below.

The first step of the algorithm is to determine a link between a commit and a defect report. A potential link is given two levels of confidence. The levels of confidence are determined by the commit information compared to the defect report (semantic confidence) and the level of data in the commit report (syntactic analysis).

Syntactic Analysis The transaction commit message is split up into a stream of tokens. Each token is one of the following:

- a *bug number*, if it matches certain syntax:
  - `bug [# \t]*[0-9]+`,
  - `pr[# \t]*[0-9]+`,
  - `show\_bug\.cgi\?id=[0-9]+`, or
  - `\[[0-9]+\]`
- a *plain number*, if it is a string of digits [0-9]+
- a *keyword*, if it matches the following regular expression:
  `fix(e[ds])?|bugs?|defects?|patch`
- a *word*, if it is a string of alphanumeric characters

Each link's initial syntactic confidence value (*syn*) is zero. It is raised by one for each of the following conditions:

1. The number is a *bug number*.
2. The log message contains a *keyword*, *or* the log message contains only *plain* or *bug numbers*.

Semantic Analysis To validate an inferred link (t, b), its information for the transactions t are checked against the defect report b. The semantic confidence value (*sem*) is based on the outcome. As with the syntactic value, *sem* starts at zero and is then raised by one whenever a condition below is met:

1. The bug b has been resolved as `FIXED` at least once.
2. The short description of the bug report b is contained in the log message of the transaction t.
3. The author of the transaction t has been assigned to the bug b.
4. One or more of the files affected by the transaction t have been attached to the bug b.

In order to choose the links that are to be used the follow conditions have to be met:

$$(sem > 1) \vee (sem = 1 \wedge syn > 0)$$

4.1.1.1 Locating Fix-Inducing Changes (Defect Insertion)

A fix inducing change is defined by Śliwerski et al. [2005] as *"a change that later gets undone by a fix"*. This means that the first change causes the software to fail, so it requires a later fix to rectify the problem. The SZZ algorithm finds fix-inducing changes by first finding the changed lines from the linked transaction and its previous transaction. The changed lines are checked to see when they were last changed. It is these changes that tell us when and where the defect could have been inserted into the program. In detail:

1. Detect *lines* L that have been touched by *change* δ in *revision* r_n. This is done by using the Git *diff* command. The diff command will highlight which file lines have been changed from one transaction to another.
2. The *annotate* command is used on the last revision (r_{n-1}). The command annotates each line with the revision that last changed the line, the author of the change and the date it was changed.
3. The output of the annotations is scanned and take for each line $s \in L$ the revision r_i that annotates line l. A revision pair (r_i, r_{n-1}) is added to set S. These revision pairs are candidates for the fix-inducing changes.
4. Set S is removed of revisions that are not possible. Those that remain are called suspects.
5. Each suspect is inspected further. The following criteria is assessed:

- A partial fix - if in suspect (r_a, r_b), r_a is a fix.
- A weak suspect - (r_a, r_b) is a weak suspect if there exists a pair (r_a, r_c) which is not a suspect.
- Hard suspect - (r_a, r_b) is a hard suspect if it is neither a partial fix or a weak suspect.

6. Revision r is *fix-inducing* if there exists a pair $(r_a, r_b) \in S$ which is not a hard suspect. Commit t is *fix-inducing* if one of its revisions is fix-inducing.

4.1.2 Enhancements and other linking tools

There have been studies which have tried to improve the bug-linking algorithm [Bird et al. 2009b, Wu et al. 2011]. The improvements they have made are described below.

4.1.2.1 Improvements by Bird et al. [2009a]

In 2009 Bird et al. [2009a] looked to improve on the bug-linking of the SZZ algorithm. Bird et al. [2009a] investigated bias within datasets; looking at whether "Are bug fixes recorded in these historical databases [Bug-linking data] a fair representation of the full population of bug fixes?". Bird et al. [2009a] based their approach on that of Śliwerski et al. [2005] but made changes in order to reduce the false negative bug-links (i.e. links that are valid references in the commit log but not recognised by the Śliwerski et al. [2005] approach). Bird et al. [2009a] relaxed the original pattern matching to find more potential mentions of bug reports in commit log messages and then verified these references more thoroughly. After scanning commit messages for numbers in the given format, these numbers were then checked to see if they were not false-positive (e.g. years or dates). The number was then checked to see if it existed in the bug tracking system. The final check was to see if the referenced bug report had a fixing activity seven days before or after the commit date. Using the same time and version constraints as Śliwerski et al. [2005], Bird et al. [2009a] were able to find 3,000 more linked bugs.

4.1.2.2 ReLink by Wu et al. [2011]

As previously outlined, traditional heuristics rely on the bug-fix committer to input the defect identification number in the commit report. This could mean that there are many false negatives in the resulting data gathered using traditional methods. Bachmann et al. [2010], Bird et al. [2009a] highlight that up to 54% of fixed bugs are not linked to bug-fix commit reports. This biased defect data could impact on the measurement of software defect prediction, as some code could be faulty, but is not labelled as such.

Wu et al. [2011] developed an automatic link recovery system, *'ReLink'*[2], to find missing links that traditional heuristics [Bird et al. 2009a, Kim et al. 2006, Śliwerski et al. 2005] do not find. ReLink uses links found by traditional heuristics to develop an acceptance criteria using machine learning. This learnt criteria is applied to the unknown links to see if it is a valid link. Wu et al. [2011] tested their method on three systems (ZXing, OpenIntents and Eclipse MAT) and manually verified a sample of their results. Overall, ReLink achieved results of around 90% precision and 70% recall, which were better than the results of the traditional heuristics alone.

[2]ReLink can be found at https://code.google.com/p/bugcenter/wiki/ReLink

ReLink has not been used to determine the bug-links in this dissertation. In the studies that were carried out using EJDT, the ReLink algorithm did not work within a sensible time period (after four days on EJDT 2.0 it had not completed). The ReLink algorithm was tested by Wu et al. [2011] on much smaller systems than those used in this dissertation. The traditional heuristic method used in this dissertation has shown that it has good recall and good precision (80% and 99% respectfully) and takes a couple of minutes to complete.

4.2 Problems with Repository Mining

Many previous studies have used defect data mined from both open and closed source system repositories (e.g. [Challagulla et al. 2005, Lessmann et al. 2008, Rahman et al. 2010, Zimmermann et al. 2009]). This mined defect data is typically used to improve our understanding about the location and nature of software defects. It is essential that the mined defect data used in these studies is good quality [Kim et al. 2011]. Poor quality data can not only invalidate the findings of an individual study, but can undermine confidence in the value of mined data.

Without the source code or the repository mining tool used by the original researcher, the quality of mined defect data can be affected by the way in which a defect mining algorithm is applied. Few defect mining algorithms can be absolutely unambiguously specified for every detail in every mining context. This means that to use an algorithm, researchers must interpret the algorithm in order to implement and then apply it. As a consequence of this interpretation, it is possible that the algorithm is applied incorrectly. This could result in incorrect defect data being collected and therefore producing invalid study results and conclusions.

In order to determine how closely the previous work using the SZZ approach could be replicated, an implementation of the SZZ algorithm was created and tested on the EJDT source code. Zimmermann et al. [2007] had previously analysed EJDT (versions 2.0, 2.1 and 3.0) and released a labeled dataset[3]. Version 3.0 was focused on to validate the SZZ algorithm implementation. A subset of data provided by Zimmermann et al. [2007] was used because the amount of data needed to be small enough to be manually validated. The EJDT package was chosen as previous research has shown this to be a frequently analysed package by different researchers [Shippey et al. 2012].

When this comparison was first undertaken, there was poor agreement between the implementation and the results by Zimmermann et al. [2007]. Zimmermann et al. [2007] had found a greater number of links. By manually inspecting the cases where the two datasets disagreed, it showed that the implementation did not correctly identify some types of transaction reports:

- Reports with just numbers located in them. E.g. 876950 59494
- Reports that have a "Fix for" prefix. E.g. Fix for 56484

This was corrected by including reports that had messages containing just numbers which also exist in the list of defect ID's. The lexer was also updated to identify tokens for the "Fix for" prefix. This formed a second implementation. In the results the first implementation is denoted as S_e and the second as S_e^+.

[3]The labeled dataset is available from http://www.st.cs.uni-slaarland.de/softevo/bug-data/eclipse/

Compared Datasets	Kappa Statistic
S_e v Zimmermann et al. [2007]	0.072
Zimmermann et al. [2007] v S_e^+	0.38
S_e v S_e^+	0.041

Table 4.1: Kappa coefficients to show the level of agreement between the implementations. The proportion of agreement is very low for all datasets.

4.2.1 A Manual Inspection

All the defect-links found using S_e and S_e^+ were manually checked and verified. This inspection was to determine if the link was a true bug-link, or if the implementations were providing incorrect links. For each bug-link found by each of the implementations, the commit message was examined to determine if it met the right criteria (as set out in the previous section). Each link was given a classification of 'True' (the bug-link is a true bug link), 'Number' (the commit message only contains a number that is in the defect number list) or 'False' (the bug-link is incorrect). For the purposes of precision and recall, a 'Number' was also classified as 'True'. This manual inspection allowed for the comparison of each of the implementations. The manual inspection would help determine if the implementations were achieving accurate results. This manual inspection could also be used in the future to test other bug-linking techniques.

4.2.2 Comparing Implementations - The Results

Both S_e and S_e^+ are able to find defect links. The second (S_e^+) discovers many more links than the first one (S_e). This is due to the problem discussed in the previous section. The number of defect-links found in EJDT by S_e , S_e^+ and Zimmermann et al. [2007] are compared in Figure 4.1. Figure 4.1 shows that S_e^+ contains more defect-links in the EJDT package than the Zimmermann et al. [2007] algorithm did. Figure 4.1 also shows that S_e^+ contains all but one of the same linked defects as Zimmermann et al. [2007]. The defect link that S_e^+ does not detect is a commit message with two defect numbers combined (8221773081). It is unknown how Zimmermann et al. [2007] decided that this was a defect link with this commit message.

S_e contains only around 10% of the same defect-links than S_e^+. S_e^+ finds just under 200 more linked-defects than SZZ. A kappa coefficient was calculated to determine the level of agreement between the different implementations. The kappa coefficient is a statistical measure of the inter-rater agreement for categorical items [Carletta 1996]. The coefficient takes into account the chance of the agreement occurring by chance. Equation 4.1 shows how the kappa coefficient is calculated, where $Pr(a)$ is the observed agreement and $Pr(e)$ is the probability of a chance agreement. Table 4.1 shows that there is a little agreement between Zimmermann et al. [2007] and S_e, and S_e and S_e^+. The best agreement is between S_e^+ and Zimmermann et al. [2007] at 0.38 (however this agreement level is considered only 'fair').

$$\kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)} \tag{4.1}$$

To investigate the discrepancies between defects mined using S_e^+ and those mined by Zimmermann

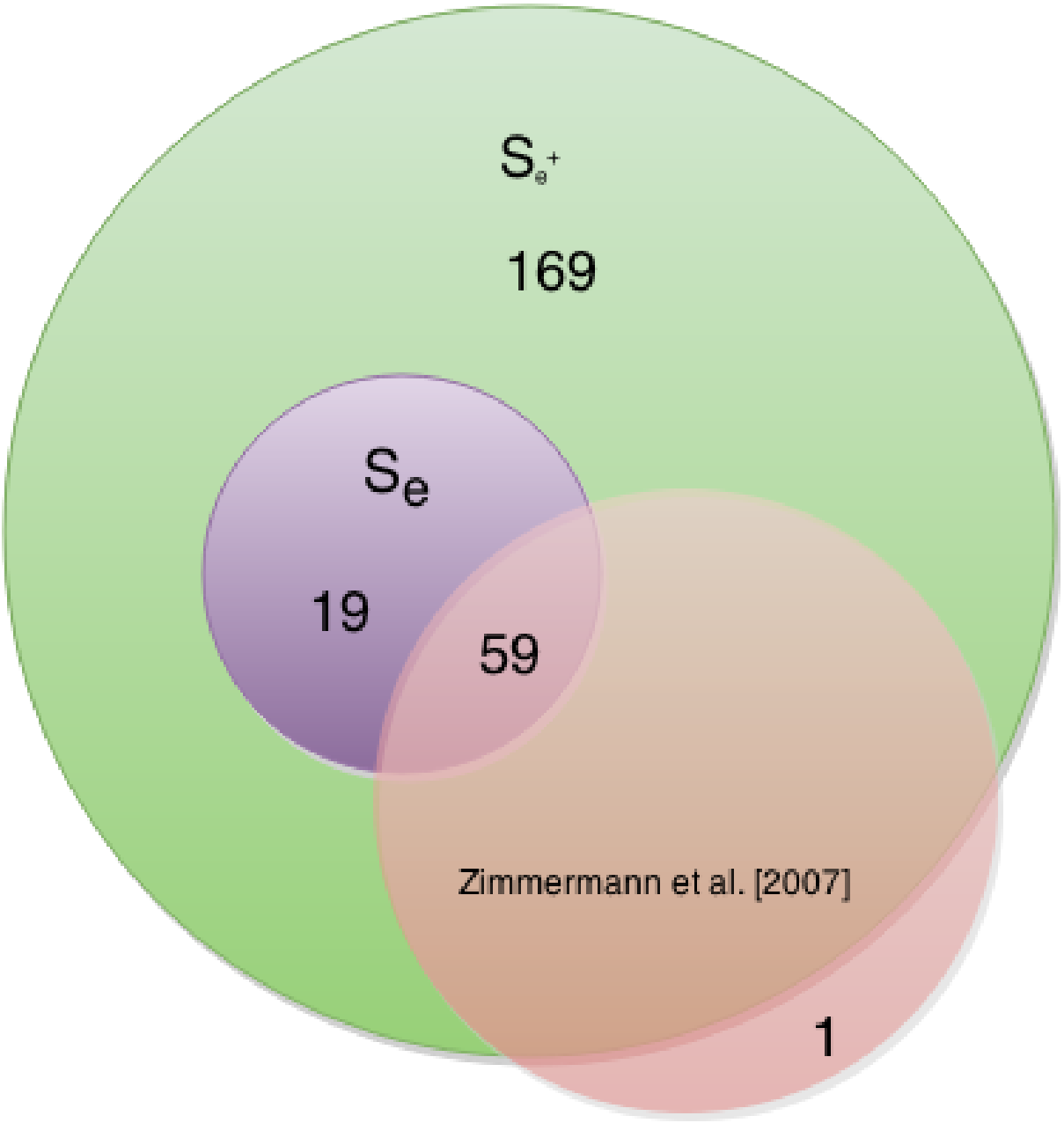


Figure 4.1: A Venn Diagram to show the agreement between the three implementations. S_e^+ includes all but one bug-link of both the other two implementations.

et al. [2007], we looked at how Zimmermann et al. [2007] had handled VCS branches. Branches are a potential important source of data variation in data mining from repositories. The use of branches is not documented in the Zimmermann et al. [2007] algorithm. When commits on different branches are analysed, there does not seem to be a pattern. Zimmermann has commits on all but two of the same branches that S_e^+ has. Of the commits on these branches our implementation gets 16 commits (Table 4.2).

Branch Name	In Zimmermann	Not in Zimmermann	Total
master	232	457	689
remotes/origin/JDK_1_5	15	0	15
remotes/origin/R2_1_maintenance	1	0	1
remotes/origin/R3_0_2_patches	3	12	15
remotes/origin/THAW_402	1	7	8
remotes/origin/THAW_452_R30x	2	2	4
Total	254	478	732

Table 4.2: Number of commits in S_e^+ compared to Zimmermann et al. [2007] on the different branches. There does not seem to be a problem with Zimmermann et al. [2007] missing branches.

	S_e^+	Zimmermann et al. [2007]
True Positive	727	483
False Positives	5	2
Total Postives	732	485
Total Negatives	151	398
Recall	80%	53%
Precision	99%	100%

Table 4.3: Checking the bug-links for false positives. Zimmermann has similar precision but lower recall.

System	Version	KLOC	Avg Line Length	Total Methods
EJDT	2.0	216	33	10,927
EJDT	3.0	292	34	13,885
EJDT	3.1	349	35	16,052
AspectJ	1.7.0	353	33	21,980
ArgoUML	0.20	273	34	12,330

Table 4.4: The five datasets analysed in this dataset.

4.2.3 Checking for False Positives

It was important to check that the extra defect insertion points identified by our implementations are not false negatives. That is that they accurately match a transaction to a bug number. Each of the 732 linked defects were manually checked to determine whether they were correct.

Table 4.3 shows that S_e^+ has a recall of 80% and precision of 99%, compared to Zimmermann et al. [2007], who has lower recall but similar precision.

4.3 Applying S_e^+ to the Different Datasets

The implementation described in Section 4.1.1 was used to find defect-fix and defect-insertion points for the systems discussed in Chapter 3.

For each of the datasets, all of the commit messages located in the their respective Git repositories were extracted. The messages from all branches of the repository were mined, including branches with both local and remote origins. These commit messages were then analysed using a custom created lexer. The lexer is designed to parse the commit message into the tokens defined by SZZ. These tokens are used to calculate relevant confidence values. A custom lexer was created because it was unclear how Śliwerski et al. [2005] had originally parsed their log messages to find the tokens. A list of bug-links was created when the confidence values had been calculated. These bug-links were then used to find the defect insertion point in each of the datasets.

4.3.1 Results

In total the implementation of SZZ found 3,339 bug-links, fixing a total of 2,120 (See Table 4.5). Defects fixed is lower than the bug-link total as multiple commits can fix the same bug. EJDT 3.1 had the greatest amount of bug-links with 1,405, closely followed by EJDT 3.0. AspectJ has the least number of bug-links with only 17, however it also has the least number of defects associated with its version number. AspectJ in fact has the best bug-fix detection rate at 87.5, this is 1.72 greater than EJDT 3.1 and 62.5 more than the lowest bug-link detection rate posted by ArgoUML. As the versions of EJDT mature, every category in Table 4.5 increases. This could indicate that the practices involving reporting bugs, fixing them and then reporting them being fixed in a standard format improved as time went on.

System	Total Bug-Links	Defects Reported	Defects Fixed	Defects Fixed %
ArgoUML 0.20	18	56	14	25.00
AspectJ 1.7.0	17	16	14	87.50
EJDT 2.0	554	930	293	31.51
EJDT 3.0	1,345	1,072	852	79.48
EJDT 3.1	1,405	1,104	947	85.78

Table 4.5: Total amount of Bug-Links for each dataset

Table 4.6 shows that EJDT 3.0 has the most changes due to fixes, around 5,000 more than AspectJ. This is because EJDT has many more defect fixes than AspectJ. A defect fix means there will be around five different changes and each of these changes on average adds five lines (Table 4.6). Figure 4.2 shows the number of lines changed in each defect fix for each of the five systems. Most of the fixes involve a few number of lines. However in each of the EJDT systems, there are fixes which involve a large number of deleted or added lines.

Using these changes the methods that are defective can be identified. Table 4.7 shows that AspectJ had the most methods, around 6,000 more than EJDT 3.1. EJDT 2.0 the least at just under 11,000. AspectJ had the least number of defects (19) in its 1.7.0 release compared to the other four datasets. This means its defect rate is also the lowest at 0.09%. The highest defect rate was in EJDT 3.1 which had around a 5% defect rate. These defect rates will be used to test for significant defective patterns in Chapter 5.

System	Defects Fixed	Total Hunk Changes	Avg Changes per Defect Fix
ArgoUML 0.20	14	245	17.50
AspectJ 1.7.0	14	67	4.79
EJDT 2.0	293	1,708	5.83
EJDT 3.0	852	5,604	6.58
EJDT 3.1	947	4,802	5.07

Table 4.6: A table to show the changes when the defects are fixed in all datasets.

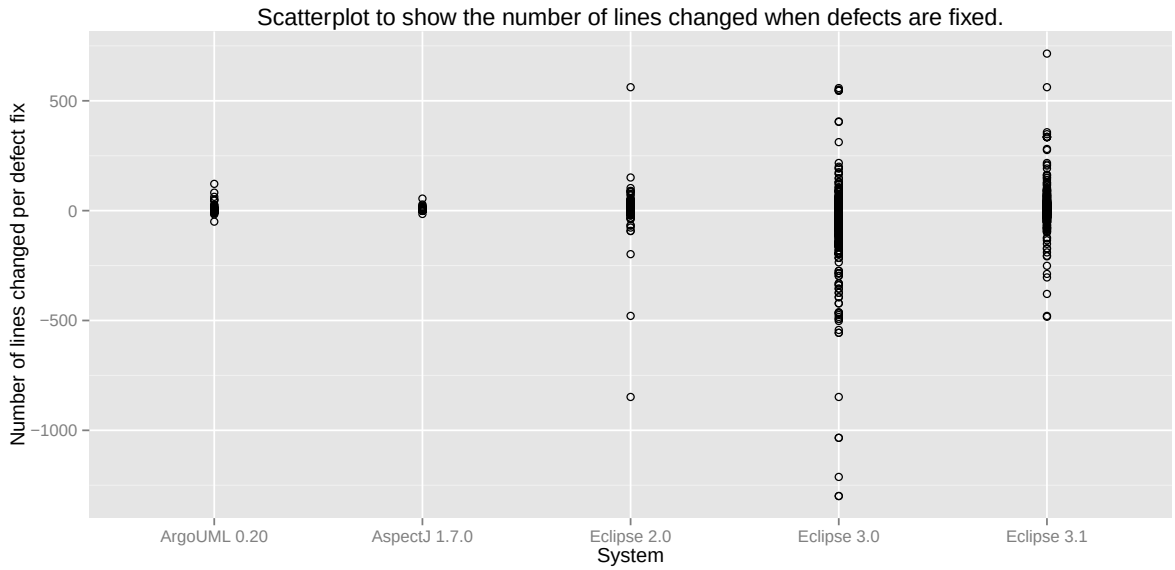


Figure 4.2: A scatterplot to show how many lines change as a defect is fixed in each system studied. EJDT 3.0 has many more lines added or deleted per defect fix than the other systems.

System	Version	Total Methods	Defective Methods	% Defective
ArgoUML	0.20	12,330	42	0.34
AspectJ	1.7.0	21,980	19	0.09
EJDT	2.0	10,927	365	3.34
EJDT	3.0	13,885	589	4.24
EJDT	3.1	16,052	797	4.97

Table 4.7: Table to show the defect density of each of the datasets.

4.4 Conclusion

This chapter has outlined how the defective methods are found in each of the datasets chosen. The defect-fix and defect-insertion points were found using an updated version of the algorithm SZZ. An implementation called S_e^+ was developed and applied to each of the five datasets. The bug links found were analysed to find defective code at the release point of each dataset. The changed code highlighted which methods were defective at each particular release.

This method was successful. It was able to find bug-links which were used to locate the defective

methods in each dataset. These defective methods will be used in Chapter 5 to find the patterns associated with defective code.

The problems encountered during the implementation of SZZ show how difficult it is to mine the same data using the 'same' algorithm. More detailed information needs to be clearly stated in papers which present an algorithm to ensure that the same results can be achieved for a dataset. In order to be confident that one can compare and synthesise results between studies all information has to be provided about how the algorithm was implemented. For example, for SZZ it is unknown how Śliwerski et al. [2005] parsed their log commits. This algorithm used its own lexer which could have resulted in the difference between the two datasets. The implementation of SZZ in this chapter suggests that Zimmermann et al. [2007]'s dataset may have too many false negatives. This could mean that subsequent studies that have used the dataset could be building sub-optimal prediction models. Kim et al. [2011] demonstrates that prediction performance decreases significantly when the dataset contains 20%-35% of both false positive and false negative noises. The results show that it is hard to be consistent in mining defect data from repositories and defect tracking systems.

CHAPTER 5

Sequencing Java Code

This chapter describes the process of transforming Java code into Java code snippets. Section 5.1 will describe how a Java piece of code is transformed into a sequence of Java programming constructs. Section 5.2 will describe how this sequence is then broken down into Java code snippets.

This dissertation introduces a new independent variable for use within (but not limited to) defect prediction. The independent variable introduced is based on Java constructs, which are to be known as *"code snippets"*. Code snippets can be short or long sequences of Java kinds. Java kinds are the tree nodes on an abstract syntax tree (AST) when the source code is parsed. In Java there are 92 different tree nodes (Tree.Kind), for example `METHOD`, `IF` or `FOR` (a full list and their meanings is available in Appendix A.1). For the rest of this dissertation, the tree nodes on the Java AST will be referred to as *kinds*. Each kind represents a different tree node for a different Java construct. A kind or a combination of kinds could represent different coding constructs such as a for loop or switch statement. One kind will often lead to other kinds, such as a `SWITCH`[1] kindmay be followed by a `CASE`[2] kind.

This chapter describes how to transform software code methods into a sequences of Java kinds. These sequences can then be broken down further into the smaller subsequences known as code snippets. Section 5.1 describes the method of how the source code is transformed into kind sequences. Section 5.2 shows how the code snippets are extracted from the kind sequences. Section 5.3 will present the data from the sequences and code snippets.

5.1 The Sequencing of Java Methods

A sequence is an syntactical representation of a piece of code, which is in the order each syntactic unit is visited when the AST is traversed. Each sequence will display the Java constructs that have been used to create the code. These sequences of constructs are ordered Java kinds, with the start and end points decided by the user: known as *restriction points*. This dissertation is examining the association of code snippets with methods, therefore the restriction point will be the start and end of a method; the `METHOD` kind. This restriction point could be any kind or a combination of kinds that are available. The Java kinds have been implemented by Oracle and are located within the AST which is created when a source file is parsed.

To extract the required method sequences from the source code the Oracle Java parser[3] is used. The parser creates a `CompilationUnit`[4] which is the AST root. Starting at the root, the AST is traversed by an edited tree walker, which is a modified `Pretty.java` class. The `Pretty.java` class is a class located

[1] A switch statement tree node. http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/tree/SwitchTree.html
[2] A tree node for a case in a switch statement. http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/tree/CaseTree.html
[3] http://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.html
[4] http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/tree/CompilationUnitTree.html

```
1  public void forMethod() {
2      int x = 10;
3      for(int i = 0; i < x; i++) {
4          System.out.println(x*i);
5      }
6  }
```

Figure 5.1: The code that is transformed in Figure 5.2 using the Pretty Printer.

```
1  METHOD;MODIFIERS;PRIMITIVE_TYPE;BLOCK;VARIABLE;
2  MODIFIERS;PRIMITIVE_TYPE;INT_LITERAL;FOR_LOOP;
3  VARIABLE;MODIFIERS;PRIMITIVE_TYPE;INT_LITERAL;
4  LESS_THAN;IDENTIFIER;IDENTIFIER;EXPRESSION_STATEMENT;
5  POSTFIX_INCREMENT;IDENTIFIER;BLOCK;EXPRESSION_STATEMENT;
6  METHOD_INVOCATION;MEMBER_SELECT;MEMBER_SELECT;
7  IDENTIFIER;MULTIPLY;IDENTIFIER;IDENTIFIER;
```

Figure 5.2: The code from figure 5.1 transformed into a `Kind` sequence. (A full list of the kinds and their meanings is located in Appendix A.)

in `Javac.Tree` which uses the visitor pattern to transform the source code into a readable format. The visitor pattern is a behavioral pattern which represents an operation that is to be performed on an element of an object's structure [Shvets et al. 2014]. The visitor pattern allows the definition of a new operation without changing the classes of the elements in which it operates [Shvets et al. 2014]. The `Pretty.java` class has been adjusted so that it traverses the AST until it reaches the desired start restriction point, it then begins to sequence the kinds until it reaches the desired end restriction point.

To sequence between the restriction points, the `Pretty.java` class will visit each kind by traversing the tree in a left depth first algorithm. This means it will visit one kind, and then visit each branch of that tree starting on the left side, until it reaches the last node. When a kind is visited, that kind is stored in an ordered array for future examination. The result is a sequence much like that of a DNA sequence. For example, Figure 5.1 shows an example of a simple for loop method. For all values of i up until 10 the method will print out $10 \times i$. When the for loop figure in Figure 5.1 is sequenced, it creates the method sequence in Figure 5.2. Figure 5.3 shows the AST which is created when this For Loop is parsed by the Java parser. Each white rectangle represents a different kind on the AST. This AST is walked to create a method sequence. The tree is walked using a left depth first algorithm (the numbers on lines represent the order in which the tree is walked) each of the kinds is placed into the sequence. Figure 5.2 shows the resulting method sequence that is created by walking the AST.

The method sequences generated by this method are stored in a MySQL database and are linked to the fault data found for the methods. This faultiness of each method was found in Chapter 4. These method sequences are used to find the Java code snippets.

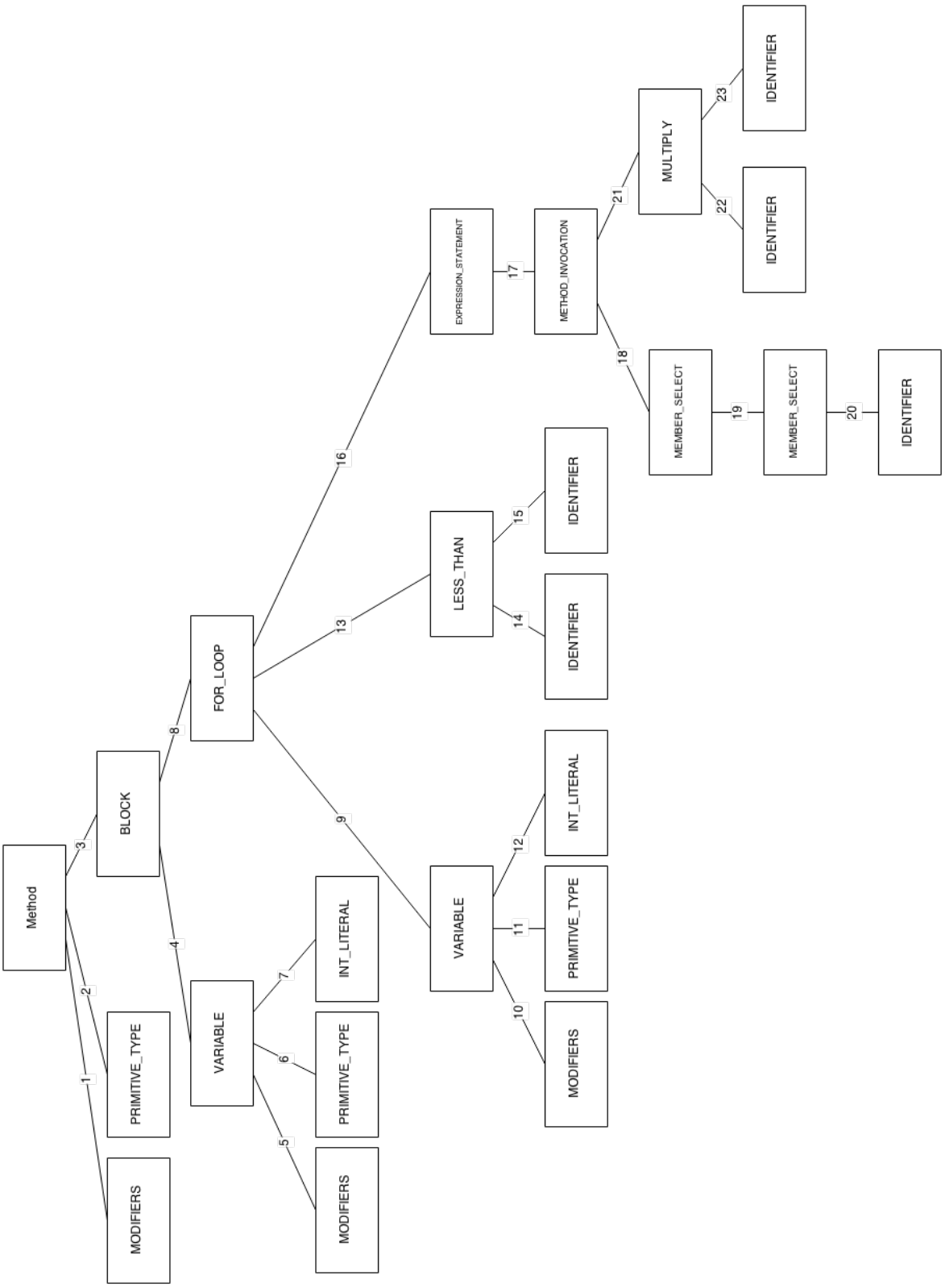


Figure 5.3: This figure shows the AST walkthrough for the `forMethod` shown in Figure 5.1. The number on the line represents the order in which the tree is traversed.

```
CS = {(A), (A; A), (A; A; C), (A), (A; C), (A; C; D), (C), (C; D), (
   C; D; E), (D), (D; E), (E)}
```

Figure 5.4: The possible set of 12 code snippets taken from an example method sequence `A; A; C; D; E`

5.2 The Code Snippets

A Java code snippet is an ordered subsequence of its method sequence. The code snippets are extracted using a sliding window approach. Consider a method sequence (M) as an array of kinds. Starting at the beginning of the method sequence, the first kind ($M[0]$) is extracted as a code snippet and placed into a set of code snippets. To extract the next code snippet, the kind ($M[1]$) is appended to $M[0]$ to create the second code snippet. This increase in code snippet length is carried out until the maximum length is reached. The maximum length can be set to any number less than or equal to the method sequence length. The algorithm starts again, but starts at the second kind ($M[1]$) and this kind is added to the database. For example, code snippets are to be extracted from an example method sequence (M) `[A; A; C; D; E]` where the maximum code snippet length is three. The algorithm starts at $M[0]$ and `A` is added to the code snippet set CS. Next, the algorithm combines $M[0]$ and $M[1]$ to create the next code snippet - `A; A`, and this is added to CS. The algorithm then combines $M[0]$, $M[1]$ and $M[2]$ to create the code snippet `A; A; C` and this is again added to CS. As the maximum code snippet length has been reached, we start the algorithm again, but this time, it will slide over by one and start at $M[1]$ and The code snippet `A` is added to CS. The next code snippet `A; C` ($M[1] + M[2]$) is added to CS. The algorithm is run through the method sequence M until no more permutations are available. In total the set CS should contain 12 code snippets. Figure 5.4 shows all the code snippets in set CS.

Figure 5.5 shows how the sliding window algorithm will extract each code snippet from the example method sequence `[A; A; C; D; E]`. The right hand side shows how the sliding window algorithm moves over the sequence to extract each code snippet. The red box shows the current code snippet to be extracted. The right hand side shows the resulting code snippet that has been extracted into the overall set. The right hand side shows the complete set of possible code snippets that can be extracted from the example method sequence (when maximum code snippet length equals three).

In this dissertation the maximum length of a snippet has been set to seven kinds in length. This is due to computational restraints in the analysis. This means there is a maximum possible total of 52 trillion possible snippets (using Equation 5.1 when $n = 7$).

The method described above was performed on all the method sequences in each of the five systems .In this dissertation the a maximum code snippet length of seven has been set. This is because as a code snippet length gets longer, their is an exponential increase in the number of code snippets available. The limit of seven kinds, limited the potential computational problems when analysing the results. The code snippets were stored into a MySQL database and mapped to their parent method sequences. They were mapped to their parent sequences in order to determine if the code snippet was in a defective method or not.

$$Permutations = \sum_{i=1}^{n} 92^i \tag{5.1}$$

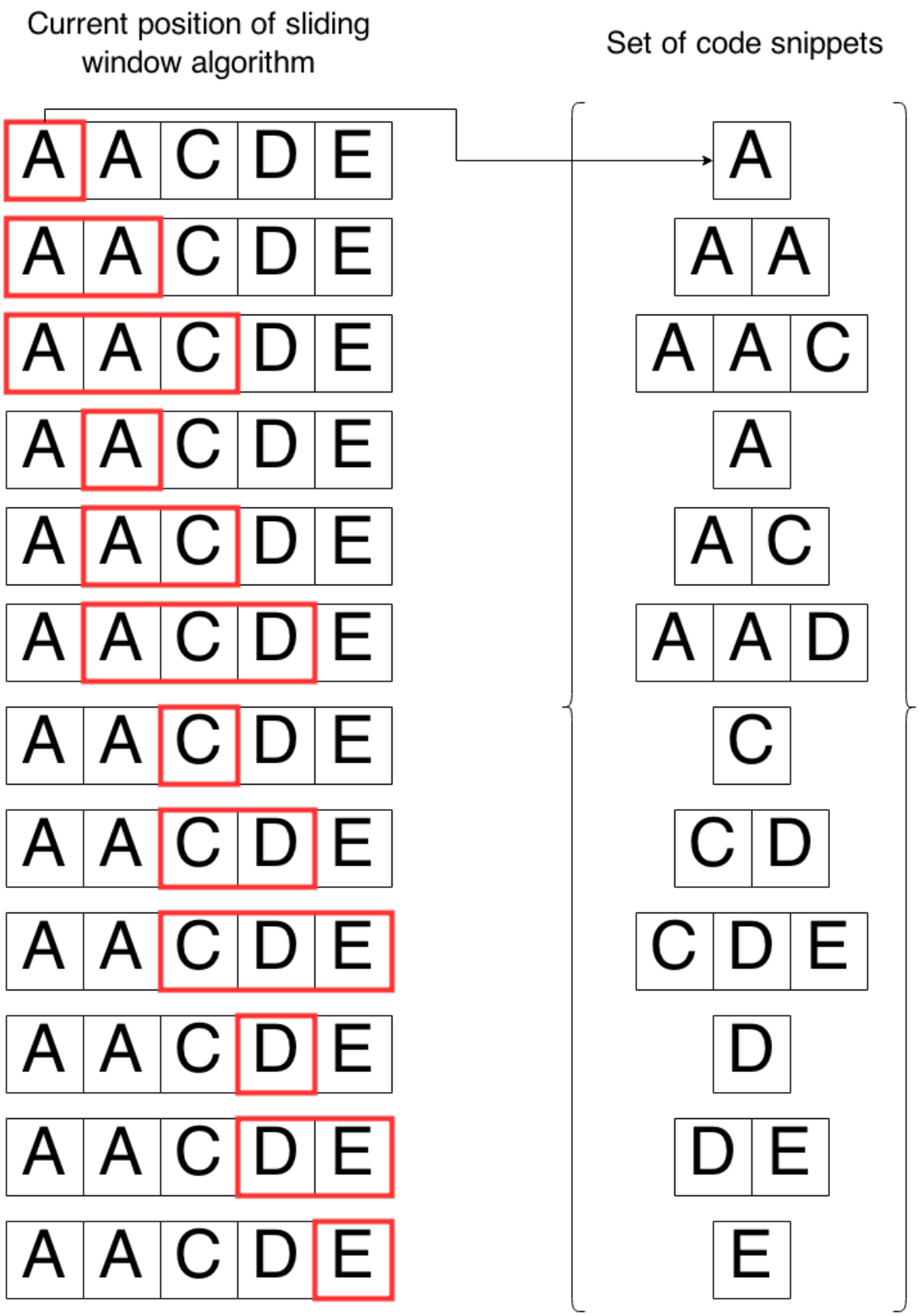


Figure 5.5: This figure shows how the sliding window algorithm extracts the code snippets from the example method sequence `A; A; C; D; E`. The right hand side shows the current position of the sliding window algorithm, while the left hand side shows the code snippet that has been extracted at that position. The right hand side is the complete set of possible code snippets that can be extracted from the example method sequence (when maximum code snippet length equals three).

System	Version	Total Sequences	Avg Sequence Length	Sequence SD	Max Sequence Length	Min Sequence Length
EJDT	3.1	16,052	59.95	153.84	6,763	1
EJDT	3.0	13,885	55.47	129.64	4,802	1
EJDT	2.0	10,927	51.21	115.06	4,410	1
ArgoUML	0.20	12,330	36.04	68.61	2,989	1
AspectJ	1.7.0	21,980	37.79	75.18	2,644	1
Total		75,174	48.09	108.47	4,321.60	1

Table 5.1: Table to show the average length of the sequences in each of the five programs. Eclipse has a longer average sequence length than the other two programs.

5.3 Results

This section will describe the data that has been gathered by the methods described above. It will describe the statistics for the sequences generated. It will then describe the descriptive statistics for the code snippets gathered from the sequences. The statistics will highlight the most popular snippets, describe some snippets that are system specific and show some snippets that are in all of the systems studied.

Table 5.1 shows the descriptive statistics for the sequences in all the systems analysed. Around 75,000 methods were sequenced across the five systems. The average length of a sequence was 48.09 kinds in length. The average length of a sequence is higher in EJDT, at around 55 kinds in length. As Eclipse matures, the average length of a sequence increases. In the time EDJT has gone from 2.0 to 3.1 the average length of a sequence has increased by around 9 kinds. The maximum sequence length is highest in EJDT and lowest in AspectJ at around 6,700 and 2,600 respectfully. Again as Eclipse matures, the average length of a sequence increases. The minimum sequence length in all of the systems is 1, which is an empty constructor. The high standard deviation for all systems highlights that there is high variability within the data. This means there are many sequences which are very low in length. The sequence lengths could not be used on their own as the variation of size between the sequences is very high. Table 5.1 shows the high standard deviation statistics for the sequence lengths in each of the systems analysed. The sequence lengths are just a proxy for the lines of code used in the module. Figure 5.6 shows a scatterplot which demonstrates the high correlation between the LOC and sequences(correlation = 0.95).

In total there are 23,502,808 code snippets extracted from the 75,000 sequences across the five systems. This includes code snippets that have been repeated in sequences, or example `IDENTIFIER, IDENTIFIER` could appear multiple times in a particular sequence. There were 351,681 unique snippets across all five systems. This is less than 1% of the total possible[5]. Table 5.2 shows the statistics for the snippets in each of the five systems. EJDT 3.1 had the most total snippets and the most total unique snippets at 6,418,917 and 208,164 respectively. EJDT 3.1 has around 110,000 more unique snippets than ArgoUML 0.20 which had the lowest total snippets and total unique snippets. As the Eclipse systems increased in age, so did the number of snippets. By EJDT 3.1 the total snippets had risen by around 42% from 3.7 million snippets in EJDT 2.0 to around 6.4 million snippets in EJDT 3.1. Total unique snippets rose by only 27% in the same period. 35,214 (10%) of the unique snippets are located in all five of the

[5]The actual percentage is $6.75 \times 10^{-7}\%$

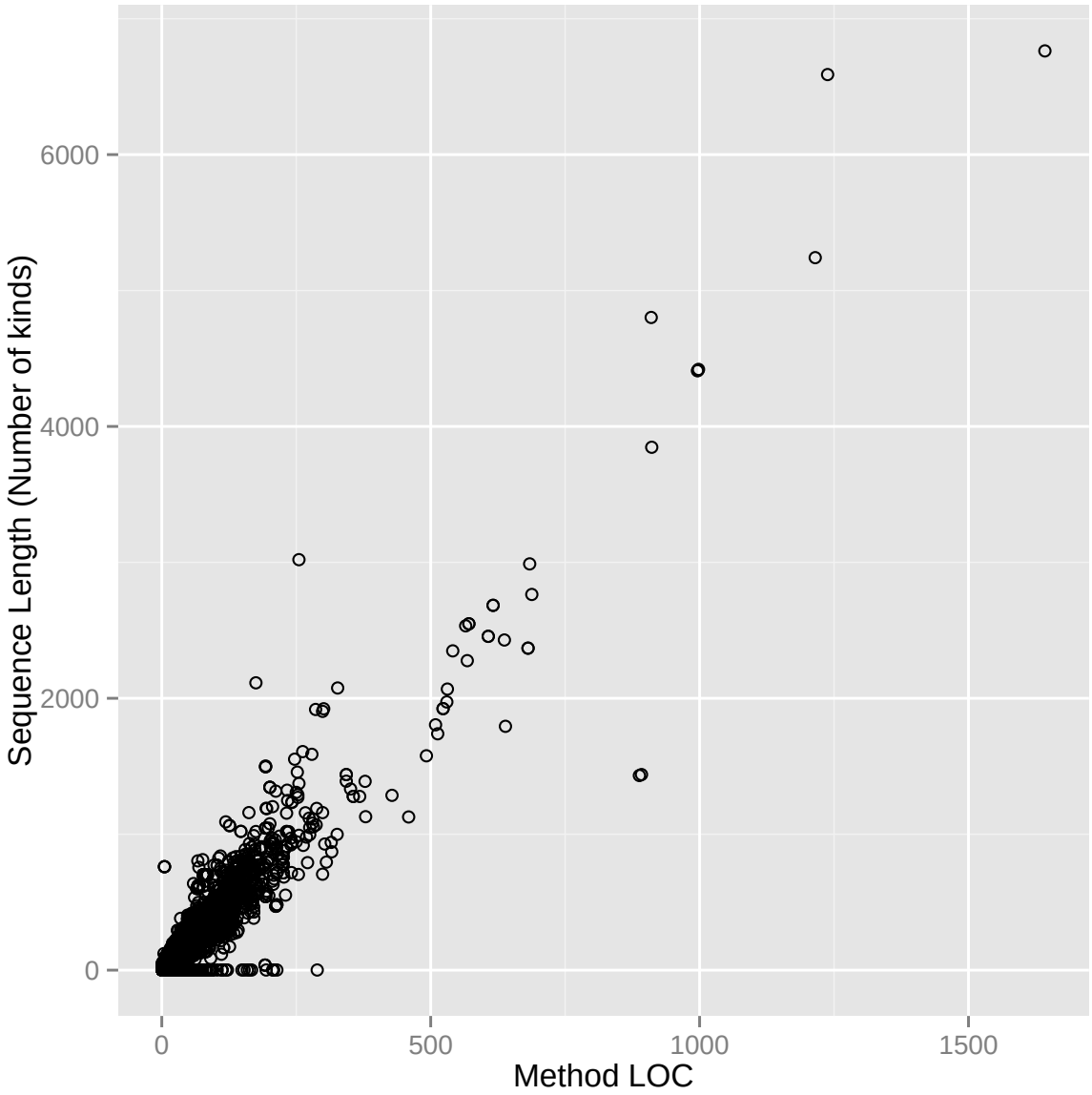


Figure 5.6: A scatterplot to show the high correlation between the LOC in a method and the number of kinds in a sequence. The calculated correlation is 0.95

systems analysed. Figure 5.8 shows a venn diagram depicting the intersections of the unique snippets between the systems. The biggest agreement is between all three of the EJDT releases, which equates to around 48,000 unique code snippets (14% of all unique code snippets).

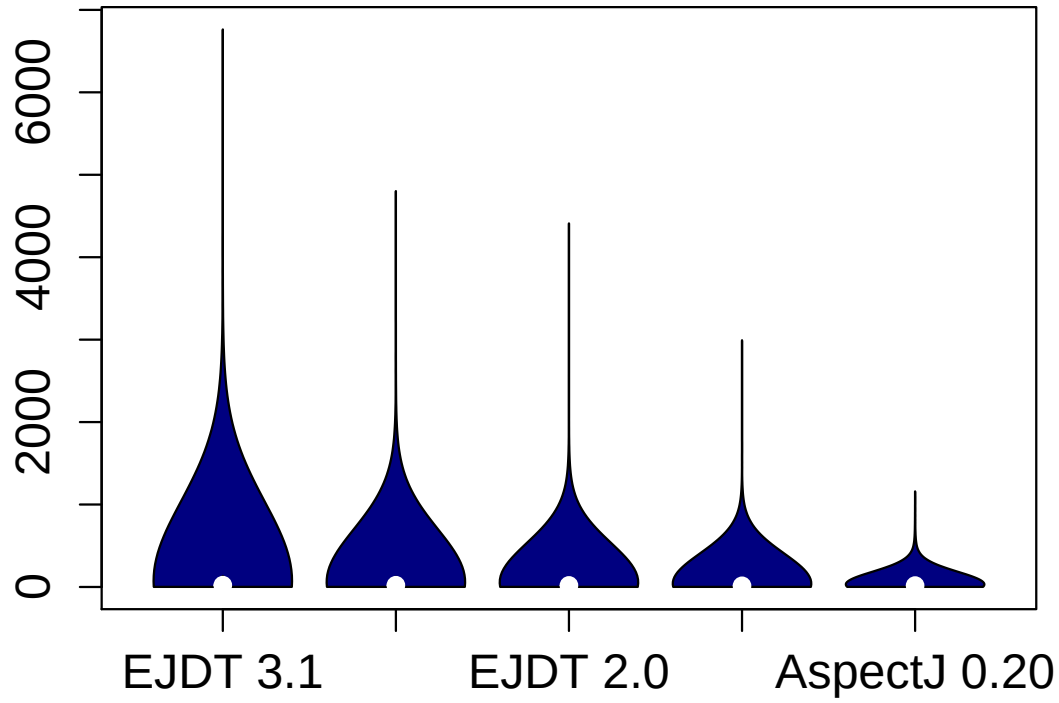


Figure 5.7: A vioplot to show the distribution of the sequence lengths in all five of the datasets analysed. There are a high proportion of sequences that are very small, compared to those sequences that are very large.

System	Total Snippets	Total Unique Snippets
ArgoUML 0.20	2,878,520	90,571
AspectJ 1.7	5,386,233	165,089
EJDT 2.0	3,702,444	150,962
EJDT 3.0	5,116,694	178,856
EJDT 3.1	6,418,917	208,164

Table 5.2: The total amount of snippets in each of the systems investigated in this dissertation.

5.4 Conclusion

This chapter focused on gathering the independent variable to be used to test the association between Java constructs and defective modules. These independent variables have been extracted by analysing the low level features of Java code via the abstract syntax tree. The Java parser has been used to sequence

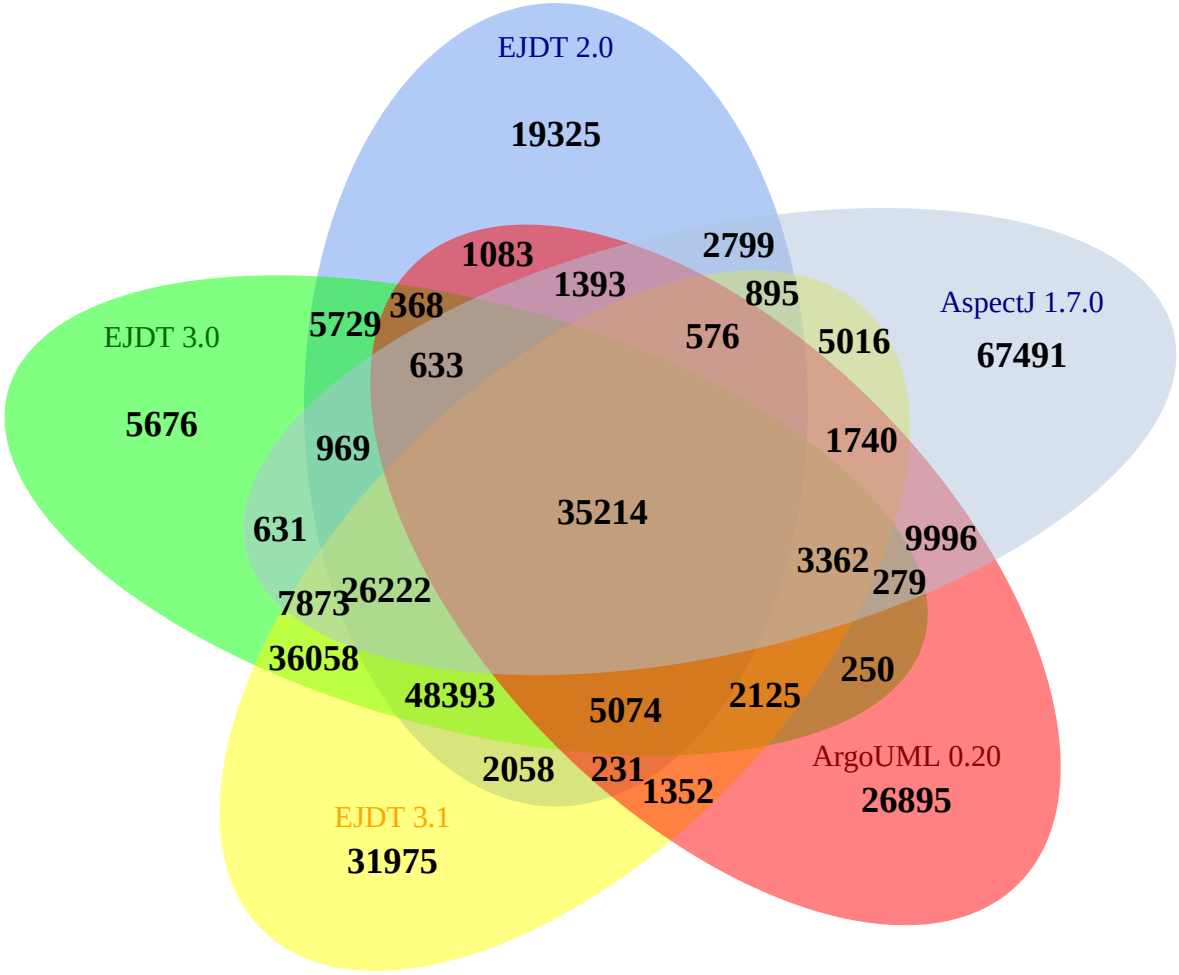


Figure 5.8: A venn diagram to show how the unique snippets found in each dataset overlap with each other. In total there are 35,214 code snippets that are found in all five datasets.

methods into sequences of Java constructs or kinds, called sequences. These sequences have been broken up to extract the Java code snippets. *Code snippets* are the new independent variables that have been introduced in this dissertation. The code snippets gathered in this chapter will be used to investigate possible associations between themselves and faultiness of methods in Chapter 6.

The code snippets discovered in this chapter are a comprehensive list of all the available snippets in the five systems. This comprehensive list will allow the analysis of more code features in defect analysis than other studies. This is because other studies usually focus on a subset of available code features (e.g. branching in McCabe [1976]'s cyclometric complexity).

The unique Java code snippets that are gathered are less than 1% of the overall available. This highlights that developers use very few of the available constructs available to them. However, there will be some constructs that cannot be used, for example a For loop proceeded directly by a method invocation (`FOR;METHOD_INVOCATION`). This finding could help reinforce the notion that programming languages are predictable and repetitive [Hindle et al. 2012]. This finding could also offer a possible empirical explanation for the relative success of recent work using genetic algorithms to automatically generate fault

fixing patches by Allamanis and Sutton [2014], Barr et al. [2014], Weimer et al. [2009] as the search space for such patches is smaller than might be expected.

Code snippets are not just limited to defect prediction and could be used in a wide range of research. For example, code snippets could be used to search for code clones within a software system. If a module has a high intersection of code snippets with another module, this could indicate a code clone could be present in these modules (e.g. Figure 2.9 shows four methods that are partial clones of each other in a southern plot type graph). Another application is to use code snippets to investigate the evolution of code within a system. They could be used to investigate whether different code snippets appear in different releases of a system, for example do certain code snippets only appear in major or minor releases?

CHAPTER 6

Determining Associations Between Faults and Java Code Snippets

This chapter will describe the method used to test the hypothesis that Java code snippets are associated with defective modules. The chapter will describe the binomial statistical test, how this is applied to the code snippets using a short example. The chapter will then present the code snippets that are significantly associated with defective modules. The results will highlight which snippets significantly associated with faults appear most frequently across the five datasets, which snippets significantly associated with faults appear in the most defective modules and which snippets always appear in defective methods.

6.1 Aim

This chapter investigates whether identifying low level features of code in the form of snippet information allows us to differentiate between faulty and non-faulty code. These features of code have been identified previously in Chapter 5 by performing a fine grained analysis of the structural building blocks of that code. The aim of this chapter is to determine if any of these lower-level features of code are associated with either faulty or non-faulty code. This chapter will determine the answers to the three research questions being investigated in this dissertation:

RQ1 Are any code snippets significantly associated with:

(a) faulty code?

(b) non-faulty code?

RQ2 Do the associations between code snippets and faultiness change as a system evolves?

6.2 The Binomial Test

This section will detail what the binomial test is, how it is used and why it was used. It will include a brief example of how it has been used to determine if a particular pattern is significantly associated with faulty code.

The association between the code snippets and faulty methods will be tested using the binomial test. The binomial test is an exact test of the statistical significance from an expected distribution of observations in two categories. The binomial test has been used as the data that is to be used fits the binomial test criteria (detailed in Chapter 3) and because the test determines if there is significant deviation from the

expected value. The binomial test will test the difference between the expected proportion of defective methods a code snippet is in, compared to the actual proportion of defective methods the code snippet is in. If there is a significant deviation between the expected proportion of defective methods the code snippet is in and the actual proportion, then the code snippet is said to be associated with faults.

The binomial test needs three pieces of information to function; the expected proportion of faulty/non-faulty methods, the number of methods a code snippet is in and the number of defective methods a code snippet is in. The expected proportion of faulty methods is calculated by dividing the total number of methods in a system with the total number of defective methods within a system. Table 4.7 shows the defect proportions for all the systems investigated in this dissertation. An example of how to perform the test for a particular code snippet is outlined below

Table 4.7 shows that in EJDT 3.0 there is a 4.24% chance of a method being faulty. In the 12,942 methods which contain the snippet `METHOD; MODIFIERS` (for example `public void foo()`) there are 565 faulty methods (4.37%). To test if 4.37% is significantly different from 4.24% we used the Binomial statistical test because our data is binary. Our null hypothesis is that $P \leq 0.0424$ and our alternate hypothesis is that $P > 0.0424$ with a 99% confidence level ($\alpha = 0.01$). It is assumed that the data follows the Binomial distribution $B(12942, 0.0424)$ so that:

$$\begin{aligned} P(X > 565|p = 0.0424) &= 1 - P(X \leq 564|p = 0.0424) \\ &= 1 - 0.7517 \\ &= 0.2483 > \alpha = 0.01 \end{aligned} \tag{6.1}$$

As 0.2483 is greater than the set significance level of 99% ($100 * (1 - \alpha)$) then there is not sufficient evidence to reject the null hypothesis and so there is no evidence that the code snippet `METHOD;MODIFIER` is associated with the faultiness of a method. The binomial test is also carried out to test if there is evidence that a snippet has an association with non faulty methods.

6.3 Results

The results section will be split into three sub sections, one for each of the three research questions. Section 6.3.1 will show that there are snippets significantly associated with faults. Section 6.3.2 will highlight that finding snippets significantly associated with non-faults is much harder. Finally, Section 6.3.3 will show how the snippets change over three releases of EJDT.

6.3.1 RQ1a - Are any code snippets significantly associated with faulty code?

There are significant code snippets in all of the five systems. Table 6.1 shows that EJDT 3.1 has the most code snippets significantly associated with faults, both by number (29,900) and by percentage (14.36%). ArgoUML has the least number of code snippets associated with faults (935) which is 1% of the total snippets. Using a binomial distribution, we would have expected only 0.5% of snippets to be significantly associated with faults at the 99% significance level. Table 5.2 shows that as the fault percentage decreases, the percentage of significant code snippets tends to decrease. EJDT 3.1 has 7% more code snippets that are significantly associated with faults than EJDT 2.0.

Table 6.2 shows the most frequently occurring snippets significantly associated with faults in each of

System	Release	Unique Snippets	Sig Snippets	%
EJDT	3.1	208,164	29,900	14
EJDT	3.0	178,856	22,723	13
EJDT	2.0	150,962	10,182	7
AspectJ	1.7.0	165,089	2,975	2
ArgoUML	0.20	90,571	935	1
Total				

Table 6.1: Table to show the number of code snippets and significant code snippets for each program. EJDT 3.1 has the most code snippets significantly located in defective methods.

the systems analysed. The `EXPRESSION_STATEMENT` kind (a complete unit of expression execution terminated by a semi-colon e.g. `aValue++;`), which appears around 37,000 times (0.55% of all the total snippets significantly associated with faults), is the most common snippet significantly associated with faults across all five systems. The `IDENTIFIER` (the name of a particular expression) and `BLOCK` (a statement block, the two curly brackets e.g. `{x+x;}`) snippets are only significantly associated with faults in the three EJDT systems, but despite this the snippets are second and third most commonly significantly associated with faults. The `IDENTIFIER` kind appears most frequently in the top ten snippets significantly associated with faults, appearing five times. The `METHOD_INVOCATION` (a method call e.g. `System.`), `EXPRESSION_STATEMENT`, `MEMBER_SELECT` (member access expression i.e. the dot that selects a particular method - print.out) kinds appear second most in the top ten snippets significantly associated with faults, all appearing three times. In the top 10% (6,672) of all snippets sorted by how often they appear, the `IDENTIFIER` kind appears in 5,586 of the snippets. The `METHOD_INVOCATION`, `EXPRESSION_STATEMENT`, `MEMBER_SELECT` appear in 2,061, 1,667 and 3,259 of the top 10% snippets significantly associated with faults respectively. Table 6.3 shows the kinds that appear the most in the top 10% of most frequently occurring snippets across all five datasets.

6.3.1.1 Variation between the Systems

There are 43,960 code snippets (12% of all snippets found) significantly associated with faults in at least one of the five systems. Of these 43,960, 63% of code snippets are significantly associated with faults in only one of the systems. Figure 6.1 shows that 0.46% (201) are significantly associated with faulty methods across all five systems. There should only be around 0.00352 ($351,682 \times (0.01)^5$) code snippets significantly associated with faults in all systems had the the association between snippets and faulty code been random. The 201 unique snippets which are significantly associated with faults in all five systems is far greater than would be expected by chance. Appendix A.2 has the full list of code snippets that are significantly associated with faults over all five of the systems.

Of these 201 snippets significantly associated with faulty methods, six snippets are short (only of length one) and so are relatively simple coding structures:

- `EXPRESSION_STATEMENT` - a complete unit of expression execution terminated by a semi-colon e.g. `aValue++;`.
- `IF` - an IF statement e.g. `if (a>10) {a++;}`.
- `PARENTHESIZED` - a node for a parenthesised expression - e.g. `(a+b)`.

Significant Code Snippet	Times occurring	Effect Size
EJDT 3.1		
IDENTIFIER	13,742	1.11
BLOCK	13,631	1.12
PRIMITIVE_TYPE	12,318	1.16
MODIFIERS; PRIMITIVE_TYPE	11,778	1.19
VARIABLE; MODIFIERS	10,859	1.31
EJDT 3.0		
IDENTIFIER	11,780	1.12
BLOCK	11,670	1.14
VARIABLE; MODIFIERS	9,456	1.30
VARIABLE	9,456	1.30
MEMBER_SELECT	8,631	1.49
EJDT 2.0		
IDENTIFIER	9,078	1.18
BLOCK	8,847	1.21
MODIFIERS; IDENTIFIER	6,327	1.54
METHOD_INVOCATION	6,138	1.67
MEMBER_SELECT	6,119	1.41
ArgoUML 0.20		
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER	6,110	1.59
EXPRESSION_STATEMENT	5,540	1.64
EXPRESSION_STATEMENT; METHOD_INVOCATION	4,688	1.82
IDENTIFIER; METHOD_INVOCATION	4,372	1.95
BLOCK; EXPRESSION_STATEMENT	4,304	1.71
AspectJ 1.7.0		
METHOD_INVOCATION; MEMBER_SELECT	11,426	1.92
EXPRESSION_STATEMENT	10,951	2.01
IDENTIFIER; IDENTIFIER	10,713	2.05
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER	10,083	2.18
EXPRESSION_STATEMENT; METHOD_INVOCATION	8,956	2.33

Table 6.2: The top five most frequently occurring snippets significantly associated with faults in each of the five systems.

Kind	Number of code snippets
IDENTIFIER	5,586
MEMBER_SELECT	3,259
METHOD_INVOCATION	2,061
EXPRESSION_STATEMENT	1,667
BLOCK	1,576
PARENTHESIZED	1,545
MODIFIERS	1,456
VARIABLE	1,416
IF	992
ASSIGNMENT	863

Table 6.3: The number of code snippets a single kind appears in the top 10% most frequently occurring code snippets significantly associated with faults across all five of the systems

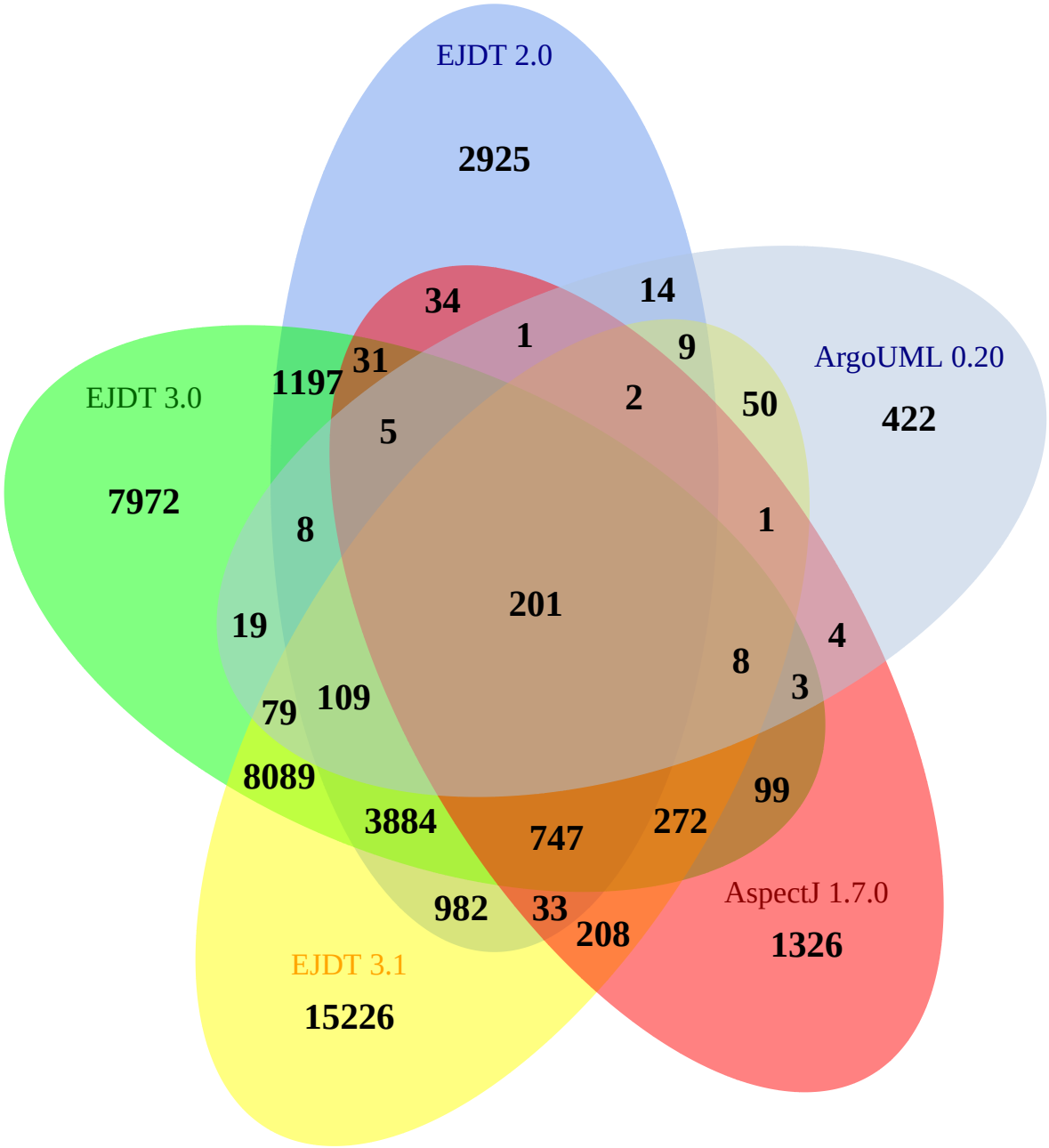


Figure 6.1: A Venn diagram to show the distribution of associations between each of the studied systems. In total 201 snippets are significantly associated with faults in all five of the systems.

- `NOT_EQUAL_TO` - the condition statement e.g. `!=`.
- `CONDITIONAL_AND` - the condition statement e.g. `&&`.
- `NULL_LITERAL` - instances representing the use of a null value e.g. `null`.

Code snippet	Total Methods	% Total Methods
METHOD_INVOCATION; MEMBER_SELECT	90	46
MEMBER_SELECT; IDENTIFIER	84	43
METHOD_INVOCATION; MEMBER_SELECT;IDENTIFIER	54	28
IDENTIFIER; IDENTIFIER	53	27
IDENTIFIER; METHOD_INVOCATION	37	19

Table 6.4: Table to show the top five sub-snippets in the 195 snippets greater than length one that are significantly associated with faults in all five datasets. `METHOD_INVOCATION;MEMBER_SELECT` appears most frequently.

```
1         public void resolve(BlockScope scope) {
2                 MethodScope methodScope = scope.methodScope();
3                 MethodBinding methodBinding;
4                 TypeBinding methodType =
5                         (methodScope.referenceContext instanceof
                             AbstractMethodDeclaration)
6                                 ? ((methodBinding = ((
                                      AbstractMethodDeclaration) methodScope.
                                      referenceContext).binding) == null
7                                           ? null
8                                           : methodBinding.returnType)
9                                 : VoidBinding;
10                if (methodType == null || methodType == VoidBinding) {
11                        scope.problemReporter().javadocUnexpectedTag(this.
                              sourceStart, this.sourceEnd);
12                }
13        }
```

Figure 6.2: The red portion of text is an example of the code snippet `METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT`. This method was faulty in the EJDT 3.0 system in class JavadocReturnStatement.java.

The remaining 195 snippets were examined and `METHOD_INVOCATION; MEMBER_SELECT` (the call of a method followed by the dot to select the method - e.g. `System.`) appears most in the 195 snippets, closely followed by `MEMBER_SELECT; IDENTIFIER` (the dot of a method call followed by the identifier of the called method - e.g. `.out`). Table 6.4 shows the top five sub-snippets in the 195 snippets greater than length one that are significantly associated with faults in all five datasets.

Figure 6.2 shows an example of a method in EJDT 3.0 that had the code snippet `METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT` within it. The red portion of text highlights the code with this code snippet. This code snippet was identified as being significantly associated with faults in all five of the systems analysed.

6.3.1.2 The Size of the Snippets

The length of the snippet was investigated to see whether snippet size had an effect on association with faulty code. A longer snippet being significantly associated with faulty code could show that specific

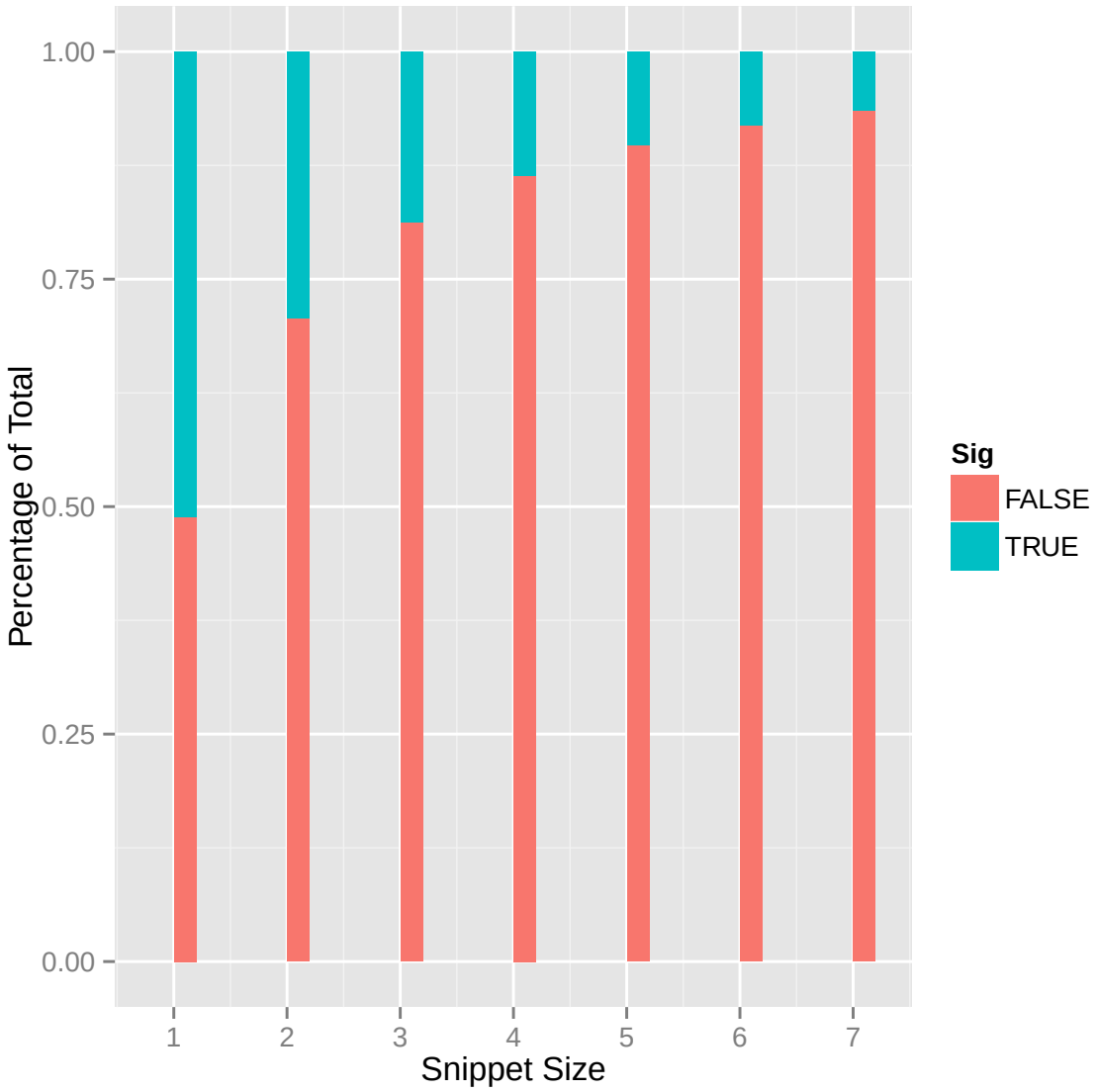


Figure 6.3: A bar chart to show the percentage of significant snippets to non significant snippets at each code snippet length. As the code snippet length increases, the chance of a typical code snippet of that length being significant decreases.

longer code constructs are potentially more harmful to a system.

Figure 6.3 is a bar chart that shows the significance levels for each of the different lengths of snippets. As the code snippet length increases, the amount that are significant decreases. When the snippet length is one, over all five of the systems, just over half of the snippets (51%) are significant, this drops to just 6% by the time the code snippet equals seven kinds in length. Figure 6.4 shows the bar charts for each system individually. All of the systems follow the same pattern, with the smaller code snippet lengths having a greater chance of being significantly associated with faults. In EJDT 3.1 and EJDT 3.0 the chance of a code snippet of length one being significantly associated with faults is just over 75%. EJDT

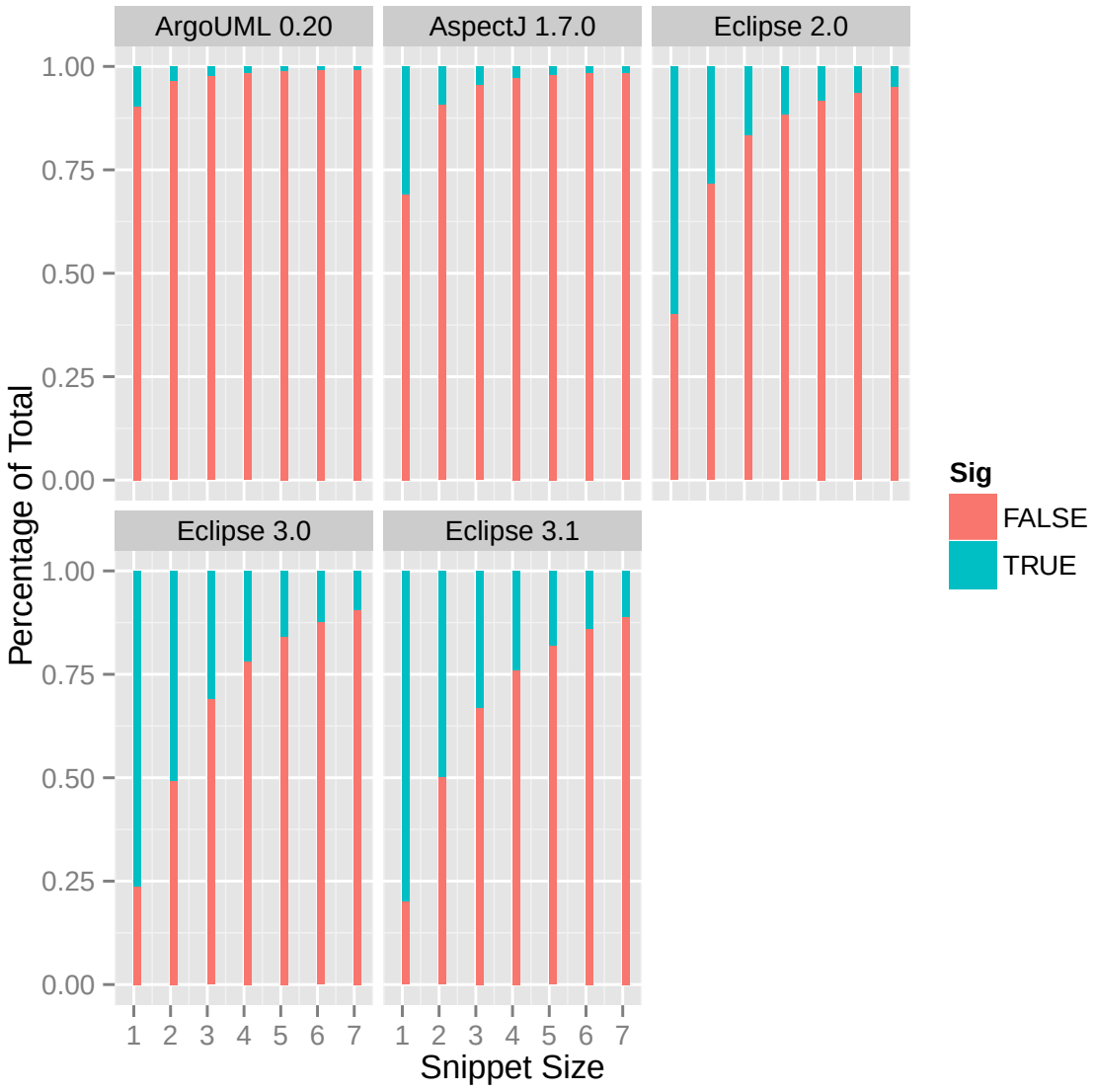


Figure 6.4: A bar chart to show the percentage of significant snippets to non significant snippets at each code snippet length for each of the five systems. EJDT 3.1 and EJDT 3.0 code snippets with a length of one have the greatest chance of being snippets significantly associated with faults.

2.0 is a little lower at just under 60%. An ArgoUML 0.20 code snippet has just over a 10% chance of being significantly associated with faults.

6.3.1.3 Those Code Snippets that are Always Faulty

In each of the systems there are snippets that are significantly associated with faults that appear only in faulty methods. Across all five of the systems, there are over 56,000 code snippets significantly associated with faults that only appear in faulty methods. However the number of methods in which

	Percentage in Methods		
	1 to 5	6 to 10	10+
ArgoUML 0.20	100.0000	0.0000	0.0000
AspectJ 1.7.0	100.0000	0.0000	0.0000
EJDT 2.0	99.2739	0.4415	0.2845
EJDT 3.0	99.5031	0.4969	0.0000
EJDT 3.1	99.3392	0.6212	0.0633
All Systems	99.3933	0.5375	0.0798

Table 6.5: Table to show the percentage of code snippets that are always significantly associated with faults and the percentage of methods they appear in. Code snippets that are always significantly associated with faults appear in one to five methods 99% of the time.

these snippets occur is very low. Table 6.5 shows that more than 99% of the snippets always associated with faults only appear in five methods or less in every system analysed. Across all five of the systems, 78% of the snippets always associated with faults appear in one method only. A snippet that is always associated with a fault appearing in 10 or more methods is extremely rare, with only EJDT 2.0 and EJDT 3.1 having such cases.

EJDT 2.0 has 29 different snippets significantly associated with faults, which appear in 10 or more methods, while EJDT 3.1 has 10. Table 6.6 shows the full list of the 29 code snippets significantly associated with faults that are always defective. EJDT 2.0's 29 code snippets are located in a very local space in the system. They appear only in two packages - `Dom` and `Codeassist` and in only 118 methods in these packages. Eight of the significant snippets appear over 50 times and exist only in three methods. On manually inspecting instances of these snippets it was suspected that code cloning (specifically type III clones[1]) created this problem. As one file contains 14 of the snippets[2] and the methods in this file all have the same structure which seems to have been changed in each of the defective methods throughout the file.

Table 6.7 shows that in the 118 methods that contain the 29 significant snippets, there are only four different types of method names; `clone`, `get*`, `find*` and `acceptPackage`. The `get*` and `find*` method names are proceeded by various names such as `getArray` or `findFields`. Figure 6.5a shows an example of one of these methods that have the snippets that are always associated with faulty methods. The code seen in Figure 6.5a is very similar to the code seen in Figure 6.5b. This similar code is repeated for nearly all the `get*` method types. These similarities and the small amount of different method types could indicate that the great number of significant snippets always being faulty could have been a type III code cloning problem. The red source code text highlights the code snippet `IDENTIFIER; NEW_CLASS; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN`. This code snippet appeared in all 52 defective methods.

Code Snippet	Total Methods	Faulty Methods	Effect Size
IDENTIFIER; NEW_CLASS; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN	52	52	29.94
IDENTIFIER; NEW_CLASS; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN; IDENTIFIER	52	52	29.94
METHOD_INVOCATION; IDENTIFIER; NEW_CLASS; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN	52	52	29.94
NEW_CLASS; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN	52	52	29.94
NEW_CLASS; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN; IDENTIFIER	52	52	29.94
IDENTIFIER; NULL_LITERAL; BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION; IDENTIFIER; NEW_CLASS	50	50	29.94
NULL_LITERAL; BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION; IDENTIFIER; NEW_CLASS	50	50	29.94
NULL_LITERAL; BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION; IDENTIFIER; NEW_CLASS; IDENTIFIER	50	50	29.94
TYPE_CAST; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; IDENTIFIER; IDENTIFIER	37	37	29.94
METHOD_INVOCATION; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; TYPE_CAST	33	33	29.94
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN	31	31	29.94
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN; IDENTIFIER	31	31	29.94
IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN; IDENTIFIER	31	31	29.94
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN	31	31	29.94
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT	26	26	29.94
MEMBER_SELECT; METHOD_INVOCATION; IDENTIFIER; IDENTIFIER; RETURN; IDENTIFIER	21	21	29.94
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; IDENTIFIER; IDENTIFIER; RETURN	21	21	29.94
METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; IDENTIFIER; IDENTIFIER; RETURN; IDENTIFIER	21	21	29.94
TYPE_CAST; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; METHOD_INVOCATION	21	21	29.94
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; EXPRESSION_STATEMENT	13	13	29.94
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION	13	13	29.94
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; EXPRESSION_STATEMENT	13	13	29.94
MODIFIERS; PRIMITIVE_TYPE; IDENTIFIER; EXPRESSION_STATEMENT; PLUS_ASSIGNMENT; IDENTIFIER; METHOD_INVOCATION	12	12	29.94
PRIMITIVE_TYPE; IDENTIFIER; EXPRESSION_STATEMENT; PLUS_ASSIGNMENT; IDENTIFIER; METHOD_INVOCATION	12	12	29.94
PRIMITIVE_TYPE; IDENTIFIER; EXPRESSION_STATEMENT; PLUS_ASSIGNMENT; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER	12	12	29.94
MEMBER_SELECT; IDENTIFIER; MINUS; IDENTIFIER; IDENTIFIER; MINUS	11	11	29.94
MEMBER_SELECT; IDENTIFIER; MINUS; IDENTIFIER; IDENTIFIER; MINUS; IDENTIFIER	11	11	29.94
IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MINUS; IDENTIFIER; IDENTIFIER	11	11	29.94
IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MINUS; IDENTIFIER; IDENTIFIER; MINUS	11	11	29.94

Table 6.6: All of the 29 code snippets that are significantly associated with faults and are always defective in EJDT 2.0.

Method Name Type	Total Methods
clone	52
get*	52
find*	13
acceptPackage	1
Total	118

Table 6.7: Table to show the different method name types in the 118 methods that contain the 29 significant snippets that are always faulty. There are only four different method name types, which could indicate that the methods were faulty clones of each other.

```
1 public Statement getBody() {
2     if (body == null) {
3             setBody(new Block(getAST()));
4         }
5     return body;
6 }
```

(a) An example of one of the methods which contains a code snippet that is always associated with faulty methods. This method `getBody()` is from the file WhileStatement.java in the dom package. This method is similar to the method `getExpression()` shown in Figure 6.5b

.

```
1 public Expression getExpression() {
2     if (expression == null) {
3         setExpression(new SimpleName(getAST()));
4         }
5         return expression;
6 }
```

(b) An example of one of the methods which contains a code snippet that is always associated with faulty methods. This method `getExpression()` is from the file WhileStatement.java in the dom package. This method is similar to the method `getBody()` shown in Figure 6.5a

Figure 6.5: An example of two of two methods that are suspected to be code clones in EJDT 2.0. The red text is code snippet `IDENTIFIER; NEW_CLASS; IDENTIFIER; METHOD_INVOCATION; IDENTIFIER; RETURN`. This code snippet appeared in all 52 defective methods.

6.3.1.4 Code snippets snippets significantly associated with faults in many methods

There are code snippets significantly associated with faults which appear in a large number of methods. These code snippets are of interest as very common code snippets significantly associated with faults with high fault densities could indicate highly defective coding structures. This information could be useful for improving recall in a defect prediction algorithm in the future.

[1] A code clone is a code portion that is identical or similar to another in a software system. They are normally due to copy and pasting from a developer [Koschke 2007]. A type III is a copy with further modifications; where statements have been changed, added or removed. [Koschke 2007]

[2] codeassist/CompletionEngine.java

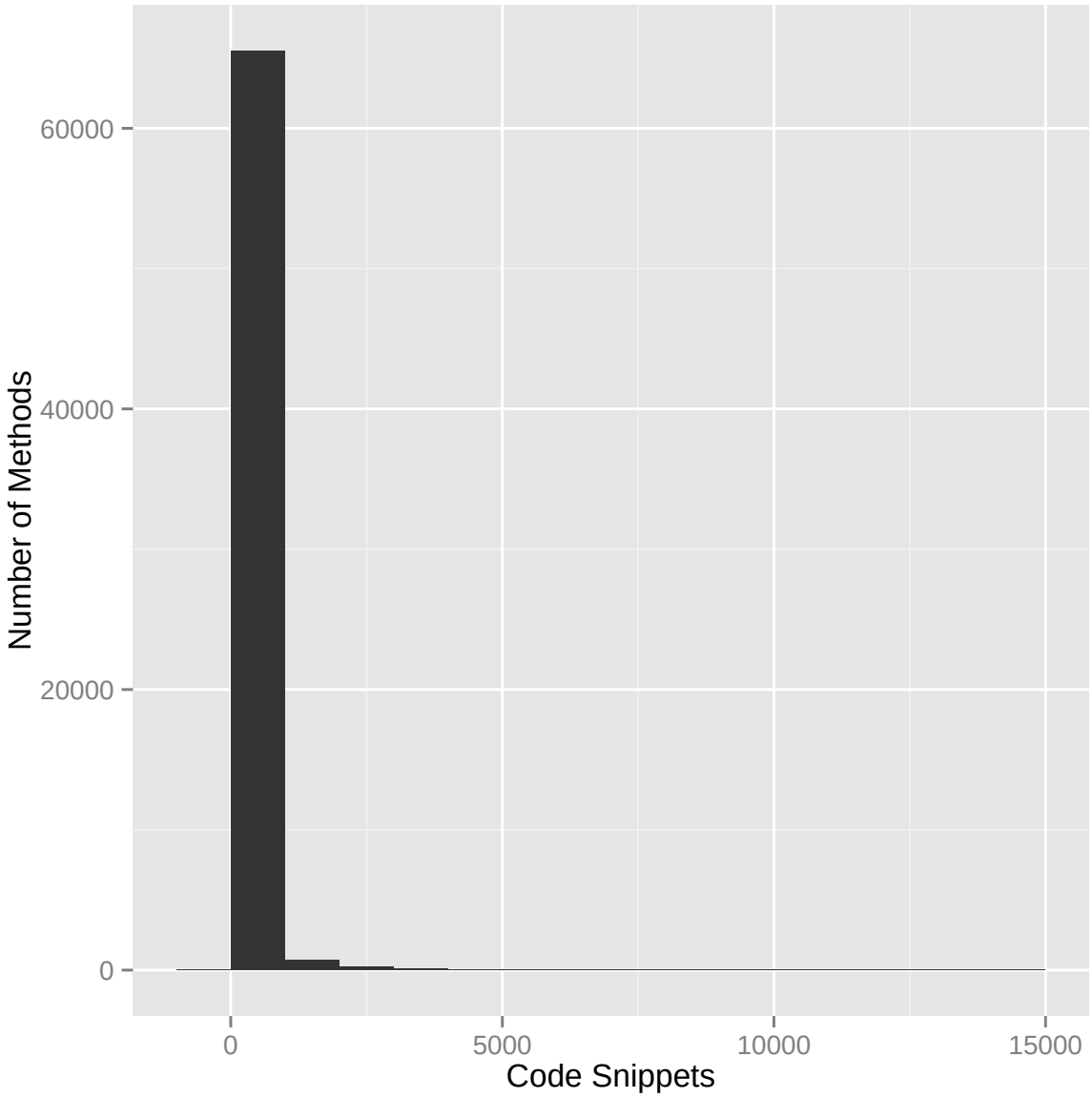


Figure 6.6: A histogram to show how the majority of code snippets significantly associated with faults are in a very small number of methods across the five systems analysed. 97% of the code snippets significantly associated with faults feature in 1,000 methods or less. 1,000 methods make up only 2% of the total methods in all of the systems. NB There are values in the far side of the x axis, however they are too small to be seen!

Figure 6.6 shows that the majority of the code snippets that are significantly associated with faults only appear in a very small number of the methods. Table 6.8 shows that 97% of the code snippets significantly associated with faults appear in 1,000 methods or less. Just over 99% of the code snippets significantly associated with faults appear in just 3,000 methods. This means that 99% of the code snippets significantly associated with faults appear in around 7% of the methods across the five systems. Table 6.9 shows the amount of code snippets that are significantly associated with faults and appear in

# Methods	Snippets	%	Cumulative %
1-1,000	42,825	97.42	97.42
1,001-2,000	550	1.25	98.67
2,001-3,000	206	0.47	99.14
3,001-4,000	90	0.2	99.34
4,001-5,000	59	0.13	99.48
5,001-6,000	49	0.11	99.59
6,001-7,000	30	0.07	99.66
7,001-8,000	19	0.04	99.7
8,001-9,000	22	0.05	99.75
9,001-10,000	22	0.05	99.8
10,000+	89	0.2	100
Total	43,960	100	100

Table 6.8: Table to show the cumulative frequencies of the code snippets significantly associated with faults and the number of methods they appear within. Just over 99% of the code snippets significantly associated with faults only feature in 3,000 methods or less. Only 380 of the possible 43,961 snippets appear in over 3,000 methods.

System	1% Method Cutoff #	Total Code Snippets
EJDT 3.1	1,230	297
EJDT 3.0	1,181	225
EJDT 2.0	800	228
Aspect J 1.7.0	1,200	228
ArgoUML 0.20	280	227
All Systems	2,640	436

Table 6.9: Table to show amount of code snippets that appear in the top 1% of methods in their respective systems. All systems total does not equal the the five systems total as a different cut off point is used for all systems.

the top 1% methods of their respected system.

Table 6.10 shows the top five code snippets (sorted by fault densities) in each system that are significantly associated with faulty methods in the top 1% of methods in their respective systems. The kinds that are most frequent in these 25 snippets are `IDENTIFIER` , `MEMBER_SELECT` and `VARIABLE` (a variable declaration e.g. `public String foo = "Foo";`). Figure 6.7 shows an example method with the code snippet `IF; PARENTHESIZED; CONDITIONAL_AND`. This code snippet has the highest fault density in EJDT 3.1. The code snippet is the start of an If statement that contains a conditional and statement. Figures 6.9 and 6.8 show code snippets highlighted within source code in ArgoUML and AspectJ respectively.

6.3.2 RQ1b - Are any code snippets significantly associated with non-faulty code?

Table 6.11 shows that there are far fewer snippets associated with non faulty methods compared to faulty methods. Table 6.11 also shows that none of these relationships are statistically significant because non of the percentages are greater than 0.5%. Figure 6.10 shows the distribution of snippets that seem related

Code Snippet	Total Methods	Faulty Methods	% Faulty	Effect Size
EJDT 3.1			4.95	
IF; PARENTHESIZED; CONDITIONAL_AND	1,345	290	21.56	4.34
IDENTIFIER; INT_LITERAL; BLOCK	1,603	338	21.09	4.25
MEMBER_SELECT; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT	1,546	324	20.96	4.22
INT_LITERAL; BLOCK	1,624	339	20.87	4.20
IDENTIFIER; BOOLEAN_LITERAL	1,373	282	20.54	4.13
EJDT 3.0			4.24	
MEMBER_SELECT; IDENTIFIER; VARIABLE	1,376	240	17.44	4.11
MEMBER_SELECT; IDENTIFIER; VARIABLE; MODIFIERS	1,376	240	17.44	4.11
MEMBER_SELECT; IDENTIFIER; IF	1,689	291	17.23	4.06
MEMBER_SELECT; IDENTIFIER; IF; PARENTHESIZED	1,689	291	17.23	4.06
PARENTHESIZED; CONDITIONAL_AND	1,205	204	16.93	3.90
EJDT 2.0			3.34	
IDENTIFIER; RETURN; IDENTIFIER	1,054	193	18.31	5.48
IDENTIFIER; METHOD_INVOCATION; IDENTIFIER	1,161	192	16.54	4.95
NULL_LITERAL; BLOCK; EXPRESSION_STATEMENT	802	120	14.96	4.16
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; METHOD_INVOCATION	902	130	14.41	4.14
IDENTIFIER; TYPE_CAST; IDENTIFIER	984	139	14.13	3.95
ArgoUML 0.20			0.34	
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION	1,061	14	1.32	3.87
IDENTIFIER; IDENTIFIER; IF	1,366	18	1.32	3.87
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED	1,366	18	1.32	3.87
IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT	1,139	14	1.23	3.61
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; BLOCK	1,056	12	1.14	3.34
AspectJ 1.7.0			0.09	
FOR_LOOP; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER	264	6	2.27	26.29
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER	310	7	2.26	26.12
IDENTIFIER; BOOLEAN_LITERAL; IF	273	6	2.2	25.43
IDENTIFIER; BOOLEAN_LITERAL; IF; PARENTHESIZED	273	6	2.2	25.43
MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER; TYPE_CAST	325	7	2.15	24.92

Table 6.10: Table to show the top five most faulty significant snippets for each of the systems that appear in the top 1% of methods ranked by percentage faulty.

```
1  private void visitIfNeeded(Initializer initializer) {
2          if (this.localDeclarationVisitor != null
3                  && (initializer.bits & ASTNode.HasLocalTypeMASK) != 0) {
4                          if (initializer.block != null) {
5                                  initializer.block.traverse(this.
                                       localDeclarationVisitor, null);
6                          }
7          }
8  }
```

Figure 6.7: The red portion of text is an example of the code snippet `IF; PARENTHESIZED; CONDITIONAL_AND`. This method was faulty in the EJDT 3.1 system in class SourceElementParser.java. This class extracts structural and reference information from a piece of source code.

```
1          private void doPendingWeaves() {
2                  for (Iterator i = pendingTypesToWeave.iterator(); i.hasNext
                       ();) {
3                          SourceTypeBinding t = (SourceTypeBinding) i.next();
4                          ContextToken tok = CompilationAndWeavingContext.
                               enteringPhase(
5                                          CompilationAndWeavingContext.
                                               WEAVING_INTERTYPE_DECLARATIONS,
                                               t.sourceName);
6                          weaveInterTypeDeclarations(t);
7                          CompilationAndWeavingContext.leavingPhase(tok);
8                  }
9                  pendingTypesToWeave.clear();
10         }
```

Figure 6.8: The red portion of text is an example of the code snippet `FOR_LOOP; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER`. This method was faulty in the AspectJ system in class AjLookupEnvironment. This class overrides the default Eclipse LookupEnvironment.

to non-faulty code (though no statistical significance could be found).

System	Release	Snippets	Sig Snippets	%
EJDT	3.1	208,164	143	0.07
EJDT	3.0	178,856	68	0.04
EJDT	2.0	150,962	140	0.09
ArgoUML	0.20	90,571	7	0.01
AspectJ	1.7.0	165,089	0	0.00

Table 6.11: Table to show the number of unique snippets significantly associated with non-faulty sequences. There are a lot fewer snippets associated with non-faulty sequences compared to faulty sequences.

```
1     public void actionPerformed(ActionEvent e) {
2         TabTaggedValuesModel model = tab.getTableModel();
3         JTable table = tab.getTable();
4         int row = table.getSelectedRow();
5         List c = new ArrayList(
6                 Model.getFacade().getTaggedValuesCollection(tab.getTarget()
                      ));
7         if ((row != -1) && (c.size() > row)) {
8             c.remove(row);
9             Model.getCoreHelper().setTaggedValues(tab.getTarget(), c);
10            model.fireTableChanged(new TableModelEvent(model));
11        }
12    }
```

Figure 6.9: The red portion of text is an example of the code snippet `EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION`. This method was faulty in the ArgoUML system in class TabTaggedValues.java. This class is table view of a UML models elements tagged values.

6.3.3 RQ2 - Does any association between code snippets and faultiness change as a system evolves?

Table 5.2 shows that as the version of EJDT increases, the number of snippets significantly associated with faults increases by around 19,000 snippets. The percentage of the snippets significantly associated with faults also increases by around eight percent.

In total across the three releases there are 42,208 unique snippets that are significantly associated with faults. Figure 6.11 shows that only 4,941 (12%) of the snippets significantly associated with faults appear in all three releases. The biggest agreement between any two of the releases is between releases 3.1 and 3.0. This is to be expected as they were released closer together. Around 62% of the total snippets significantly associated with faults appear in only one of the datasets.

The top five most common faulty significantly snippets for each of the EJDT datasets is different depending on the dataset. Table 6.10 shows that not one code snippet from one release appears in another release.

The significance percentage levels for the differing code snippet sizes differ slightly. In EJDT 2.0 a code snippet of size one has around 60% chance of being faulty, this percentage rises to over 75% in EJDT 3.1. The chance of a code snippet being faulty decreases as the code snippet size increases across all three of the releases. However, the chance of code snippet of size seven is more likely to be faulty in a more mature release of EJDT.

6.4 Conclusion

This chapter has introduced how the code snippets can be tested to determine if they are snippets significantly associated with faults. The chapter has explained the binomial test and how this is used to determine whether a code snippet is significantly associated with faults or not. The chapter then presents

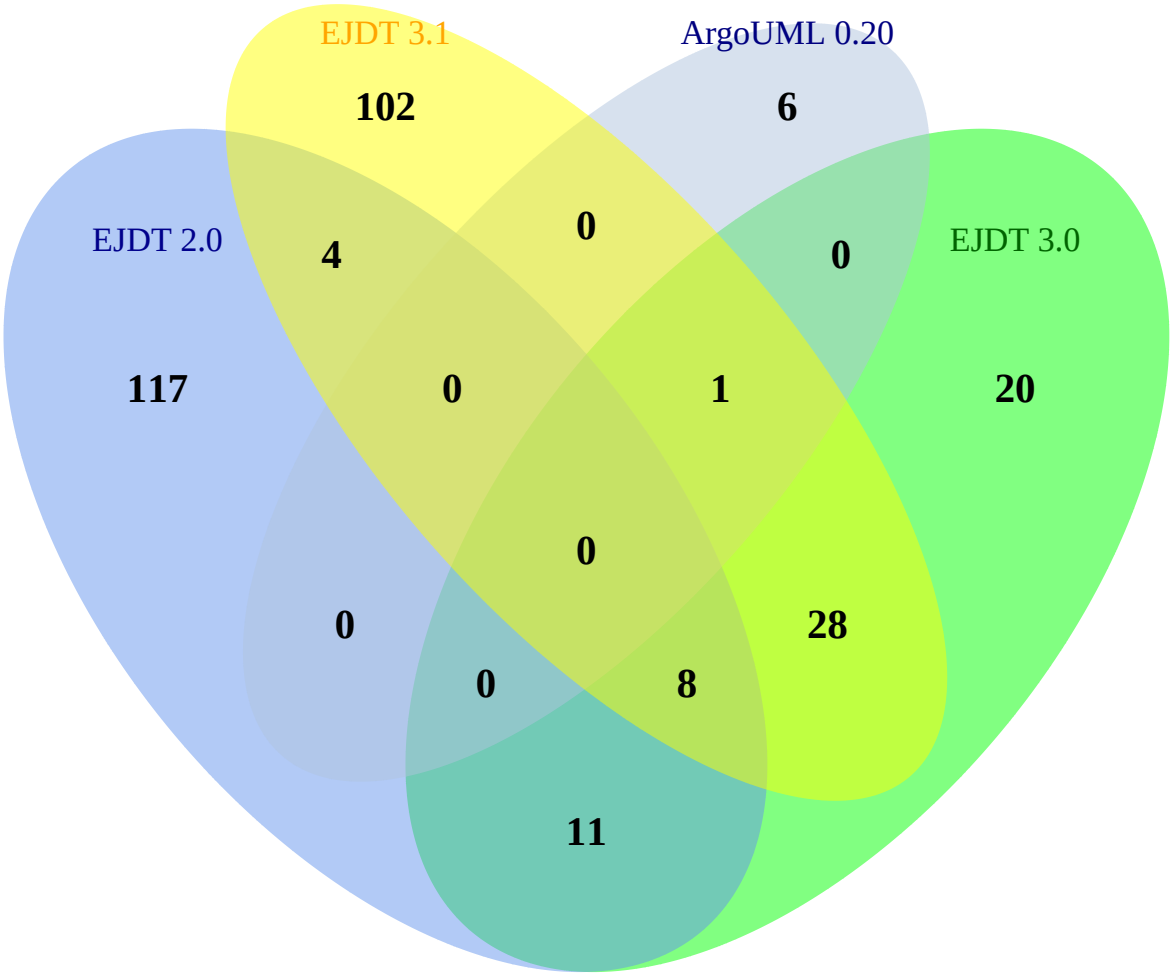


Figure 6.10: A Venn diagram to show the distribution of snippet associations between each of the systems.

the results for each of the research questions answered in this dissertation. The results presented that there are code snippets significantly associated with faults in each of the five systems examined. These code snippets range from one kind in length to the maximum seven kinds in length.

The chapter has shown that there are thousands of snippets of Java code (from the 52 trillion possible we

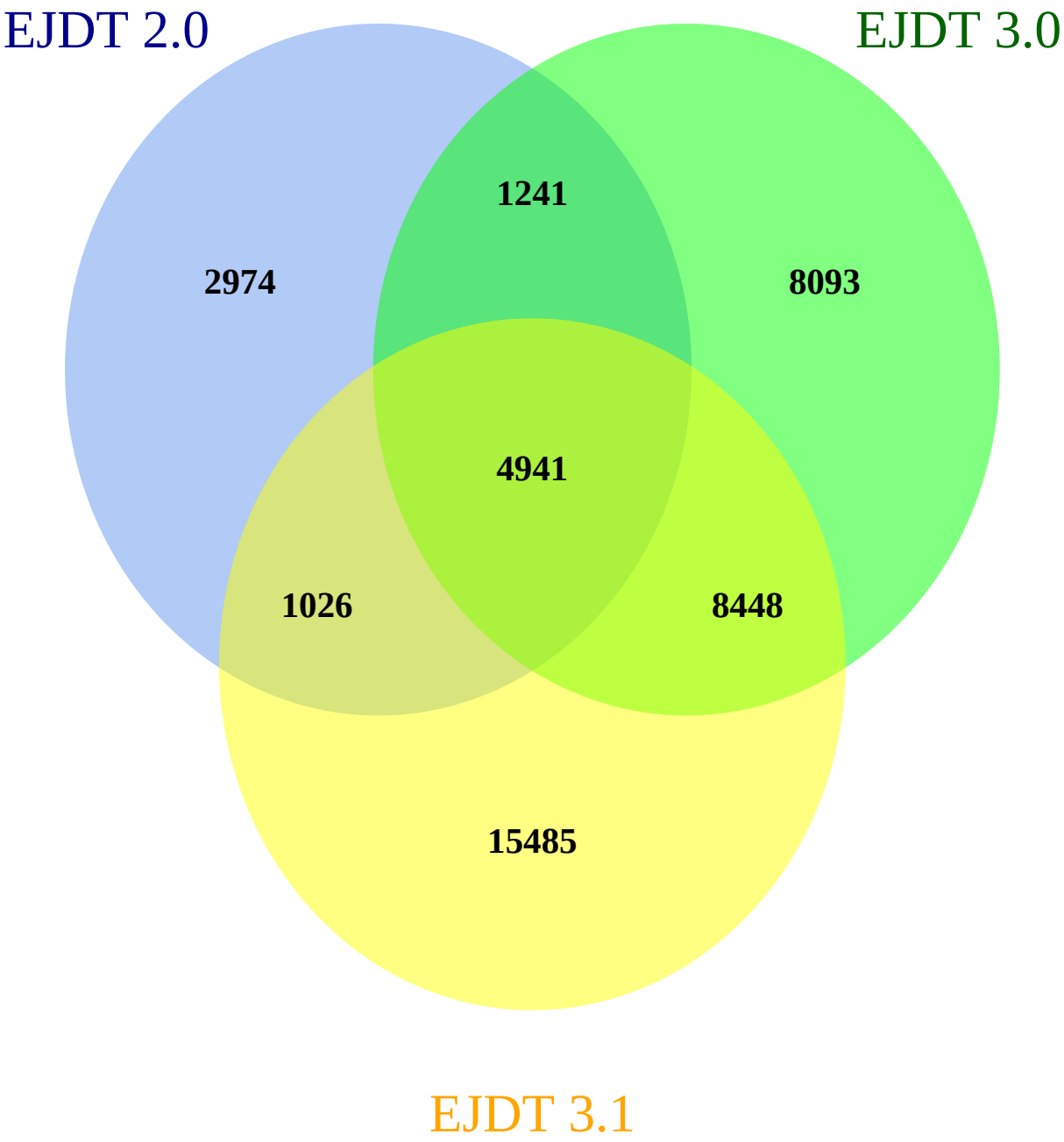


Figure 6.11: A Venn Diagram to show the association of snippets significantly associated with faulty methods among the EJDT releases. Only 12% of snippets are significantly associated with faults in all three releases.

studied) that are significantly associated with faulty code. These snippets range from one to seven kinds in length, showing that problematic snippets could be small. The snippets significantly associated with faults tend to be local to specific systems, but can also be global. Our results show that 201 snippets are located in all five of systems analysed. Although this set is small, finding any defect indicators that

transfer successfully from one system to another is rare [Zimmermann et al. 2009].

The approach is not successful at identifying code snippets that significantly associated with non-faulty code. This could indicate that it is hard to develop code that has no chance of being faulty.

Across the three versions of EJDT, there are snippets that are significantly associated with faults in all three versions. This means that snippets of code in an earlier release could be used to indicate defectiveness in later releases. However, there are many snippets that are not global across the three releases which could indicate that faults evolve over time and that particular snippets are found to be faulty and removed as the system matures. The chance of a longer code snippet being defective increases as the releases of EJDT increase. This could indicate that code is becoming more complicated as the system matures.

The results are important as they provide new evidence and information on coding structures which are fault-prone. The approach also provides a new code feature analysis technique that is relatively easy and quick to apply to systems. Companies spend large amounts of budget on finding and fixing defects. Using the findings in this chapter to reliably identify even a few extra defects early in development could save significant amounts of resources.

Chapter 7

Threats to Validity

It is accepted that there are threats to validity in the research presented in this dissertation. This chapter identifies threats from the three common threats to validity categories - internal, external and construct.

7.1 Internal

An internal threat is the small number of faults reported for both ArgoUML and AspectJ. These small datasets create the problem that it may be difficult to generate statistically significant results. It is always difficult to generate significant results with small sets of data. Small numbers of faults is typical of many software systems and it is a particular strength that the technique is able to find significant results in the systems despite a small number of faults. The technique that is shown does find significant results and also finds some of the same significant snippets as those found in the system with the larger reported faults.

The processing power available meant that the maximum length of a snippet had to be limited. Limiting the length of a snippet to seven means that possible significant snippets over this length have not been investigated. There could be frequently occurring longer snippets that are significantly associated with faulty or non-faulty code. This dissertation's research has shown that as a snippet length gets longer, the chance of that snippet being significantly associated with faults gets lower. This may mean that the results may not have missed out that many possible code snippets significantly associated with faults over seven kinds in length. With greater processing power, longer snippets could be analysed. The aim of this dissertation was to discover if there are code snippets that could be significantly associated with faulty modules, which has been achieved. Greater processing power may not have to be utilised if an investigation into the kinds that are not relevant or by not ordering the snippets. Reducing the number of snippets or the combinations of snippets will decrease the search space and improve the performance of the algorithm. This could be undertaken as future research. This was not performed in this dissertation as it was important to not bias the outcome by removing certain kinds or removing the constructs. The aim of the dissertation was to find sequences of code snippets that could indicate associations with faulty code.

7.2 External

An external threat is that the technique shown in this dissertation may only apply for the systems or the programming language studied. The results gathered from these systems may not be generalised to other systems that may be used. The aim of this dissertation was to identify possible coding constructs that are associated with faulty code, which has been achieved. The technique used to perform this should be

usable on other systems or another programming language with an AST. The technique is not system dependent and requires only the use of a source code repository and defect data. The technique has been able to produce significant results for all the systems analysed and should produce similar significant results for other systems.

7.3 Construct

Repository mining to find faulty code in systems is an inexact science. Latent defects may have not been discovered and lie dormant in the code base. Faults may also have been fixed but not reported properly in the commit message. SZZ relies on good commit messages and the technique may have not been able to find all the possible fault fix points. A manual inspection was performed to investigate if our implementation of SZZ is accurate. The implementation of the defect linking algorithm SZZ introduced in this dissertation is able to achieve both high recall and high precision. The defect insertion point of the SZZ may not have found the actual defect insertion point. This is a difficult issue for the SZZ algorithm. It is hard to distinguish between the potential defect introductions without knowing fully the motive for the change. For example, the change could have been a simple refactoring, rather than a change that caused a defect. However, the insertion point that is found does give a point in time in which the code was defective, whether a defect was introduced at that point or not. This means that while the results may contain false negatives, it will not be any worse than what is reported in this paper.

The use of Git may have mean some of the defect links or insertion points may not have the correct dates. This is because Git is a distributed versioning system. This means that each developer using Git to develop the system has its own local repository that it will control. Defect fixes or insertion points could have been made locally and not pushed to the central repository for a while after these changes have taken place. The insertion point or defect fix would be the point the developers local repository was committed to the main central repository. Although this will mean that the insertion points and defect fix may have happened sooner, these fixes would not have been available to build the distributed system until they were committed to the central repository. Therefore, for the main build of the system, where a developer commits his local changes to the central repository would be where the defect fix or insertion point is for the main build of system.

The technique described in this dissertation has focused on fault granularity at method level. This association could mean that there are code snippets within a method that are being labelled as faulty, but are not actually the cause of the fault. This could skew the results in favour of code snippets that are in frequently occurring methods. The statistical test chosen helps to negate this limitation as the code snippet has to be in significantly more of the defective methods than it is expected to. Also, a code snippet could be missed as it is not located within a method. This code snippet could be a code structure outside a method, such as in the fields of a Java class.

Chapter 8

Discussion

This chapter will discuss the findings from this dissertation's results. The chapter is divided into two main sections. Section 8.1 will discuss the answers to the research questions. Section 8.2 will discuss how the the results of this dissertation impact other areas of software engineering. The results have implications in other various areas, such software development, software testing and code cloning.

8.1 Answering the Research Questions

This section will specifically answer the research questions of this dissertation. Each research question will be discussed in detail on how these results relate to the current software engineering research.

8.1.1 RQ1a: Are any code snippets significantly associated with faulty code?

Each of the five systems (EJDT 3.1, EJDT 3.0, EJDT 2.0, ArgoUML 0.20 and Aspect 1.7.0) analysed had code snippets significantly associated with faults. There are around 65,000 code snippets significantly associated with faults. These code snippets that significantly associated with faults range in size from one to seven kinds. The one kind code snippets significantly associated with faults could be small coding constructs, for example an identifier (`IDENTIFIER`) or a null value (`NULL_LITERAL`). However, some of these small code snippets significantly associated with faults could actually be a large code construct, for example a For Loop (`FOR_LOOP`) or a Do While Loop (`DO_WHILE_LOOP`). There are many different code snippets that are significantly associated with faults that are longer than length one. The code snippets that are longer than length one have common patterns, such as a large number of method calls `METHOD_INVOCATION; MEMBER_SELECT` and conditional decision logic (e.g. `IF; PARENTHESIZED; CONDITIONAL_AND` or `IF; PARENTHESIZED; NOT_EQUAL_TOO`. Less frequently occurring code snippets significantly associated with faults included a null value followed by the start of an If statement (`PRIMITIVE_TYPE; NULL_LITERAL; IF; PARENTHESIZED`) or a While Loop with an array access being equal to some value in its arguments (`WHILE_LOOP; PARENTHESIZED; EQUAL_TO; ARRAY_ACCESS`). Due to the high volume of code snippets significantly associated with faults, it may be need to reduce the number for effective use within defect prediction. Further work is required to undergo the analysis to determine which code snippets are the most effective at making defect prediction models better. Certain code snippets could be better at differentiating between modules that, with the current metrics available, are seen as identical. This work is beyond the scope of this dissertation as it was the aim to determine which Java code constructs are associated with faults.

Less than 1% (201) of these code snippets are significantly associated with faults across all five systems. The 201 code snippets included six code snippets that were only of length one. These six included constructs such as an unit of expression (`EXPRESSION_STATEMENT`) and the not equal to logical statement

(NO_EQUAL_TO). The remaining 195 were on the majority a variety of method calls and if statements. This suggests that finding similar faulty coding constructs that are the same across systems is very difficult. This is a problem for defect prediction as it means that models using the code snippets, may not make good predictors for another system. This is not a new problem within software engineering. Previous studies have reported that defect prediction models perform poorly when transferred across systems [Bell et al. 2006, Kitchenham et al. 2007, Nagappan et al. 2006, Turhan et al. 2009, Zimmermann et al. 2009]. Bell et al. [2006] report that when they used a LOC model based on a previous study ([Ostrand et al. 2005]), on a new system, it performed much poorer at predicting defects. The result led Bell et al. [2006] to conclude that "one should not blindly apply the best model from one system to a new system". Zimmermann et al. [2009] investigated the effectiveness of using cross-project defect data to predict defects for another system. Zimmermann et al. [2009]'s results indicated that cross-project defect prediction was very difficult. Out of 622 cross-project predictions, only 22 (3.4%) actually worked. Kitchenham et al. [2007] reviewed studies that checked if a company used data imported from another company to build local effort estimation models. Kitchenham et al. [2007]'s results showed that out of seven studies, four of the effort models were worse than using local data alone. Turhan et al. [2009] investigated whether an external company's static code metrics could be used to create localised defect prediction models. Turhan et al. [2009]'s results showed that using the external company's static code metrics created a high probability of false alarms. A high number of false alarms means that a great deal of time and effort may be wasted on source code where defects are not present. Turhan et al. [2009]'s conclusion was that cross-program metrics were of little use for local datasets. Nagappan et al. [2006] analysed the defect history of five Microsoft systems and discovered that there was not a single set of metrics that were associated with faults in all five systems. Nagappan et al. [2006] also concluded that the prediction models created were only accurate when obtained from the same system. Menzies et al. [2011] suggests that one of the reasons that is causing this problem is data heterogeneity. Data heterogeneity means that global rules are not always useful for local systems. Menzies et al. [2011] analysed treatments (changes to code to improve a quality measure) from local data and global data and found that treatments from local data were superior to the global treatments. This previous work highlights the difficulties in finding cross-program predictors and why the code snippet technique method may have found only a few constructs that are significantly associated with faults across all five systems. These differences could be explained by the different characteristics of each of the systems. For example, the systems differ in terms of domain; EJDT contains the core modules of an IDE, while ArgoUML is a UML tool. This means that the code used to develop the system may differ to incorporate the different needs of the system. Similarly, the source code may be developed differently due to the different developers working on each of the products. Another possible reason for the differences in the code snippets significantly associated with faults is the development model each of the systems adopts. ArgoUML has adopted a more 'cathedral'[1] approach to develop the software where only one developer can commit the changes for the system. People can submit potential changes to this 'gatekeeper' but, they cannot directly commit changes. The 'gatekeeper' may prefer a particular way of coding and as such will reject or modify potential suggestions. EJDT and AspectJ have a more 'bazaar'[2] approach where many different developers can commit changes. These developers have to be approved, but each may have a different way of coding and as such different code nuances are placed into the systems. Each of the different system characteristics could increase the chance of the source code displaying different features.

[1]The cathedral software development approach was coined by Raymond [1999]. It is where the software code is available for each release, however it is maintained by an exclusive group of developers.

[2]A bazaar software engineering development approach is one in which the code is viewed openly by the public and can be changed or added to by a wide range of developers [Raymond 1999]

8.1.1.1 The Use of Code Snippets in Defect Prediction Models

The number of code snippets found to be significantly associated with faults across all the systems is small. Finding coding constructs that are significantly associated with faults across systems is very rare [Nagappan et al. 2006, Zimmermann et al. 2009]. As just seen, Nagappan et al. [2006] could not find any set of metrics that were associated with faults in all five of their studied systems. This dissertation has shown that there are coding constructs significantly associated with faults across all five systems. There are 201 code snippets that are significantly associated with faults across all five of the systems. This accounts for around 1% of the total code snippets. This may be a small number, but it is much higher than statistically expected. Had the association between faulty code and code snippets been random, around 0.352 code snippets would have been expected. This finding means that the 201 code snippets could be common coding structures that are frequently developed incorrectly, regardless of who the developers are and what system they are working on. The finding provides some empirical confirmation that there are Java code constructs that are fault prone across different systems. The 201 code snippets could be used to create a model that will predict defects in other systems. This defect prediction model could help improve the predictive performance of cross-system models as it is based on empirically tested coding constructs that have been shown to be defective across systems. However, as with code snippets overall, the 201 coding constructs may have to be further assessed to determine which ones have the best predictive power. Some of the constructs may not be as good as others at determining defects in other systems. They might restrict the predictive power of the model as the construct may not appear often or its effect size may be small. Investigating the performance of defect prediction models based on the 201 code snippets significantly associated with faults is outside the scope of this work and constitutes future work.

8.1.2 RQ1b: Are any code snippets significantly associated with non-faulty code?

This dissertation could find no significant association between code snippets and non-faulty code. This is because the number of significant snippets found was fewer than what should be found randomly at the 99% significance level. This could be seen as surprising as one would assume that there should be some code that is never found in faulty code. However, the result means that there is no code snippet that is significantly associated with non-defective code. It does not mean that there are no code snippets which are never defective, just that there is no statistical evidence that there are code snippets that could signal good code. This is an interesting result as, although there is evidence of the impact of poor coding styles (e.g. using message chains), there is no evidence to support the active use of good coding practices. One implication of this finding is that development practices should be based around defect-avoidance and defensive coding practices are important. There is no evidence in support of developing patterns for known fault-free code.

There are several reasons why there were no code snippets significantly associated with non-faulty code. One is that there is a very large amount of non-faulty code within the software system. For a code snippet to be significantly associated with non-faulty code, it will require a great number of code snippets to exceed the expected amount of non-faulty methods. For example, for a code snippet to be significantly associated with non-defective code in EJDT 2.0, it would need to appear in significantly more than 96.64% of non-defective methods in a particular sample. If the sample was 100 methods, then it would need to appear over 100 times in that sample, which is impossible. This indicates that finding significant results for non-faulty code snippets is going to be very rare due to the statistical properties of the data.

Unlike faulty code, non-faulty code is unlikely to cause a method to become non-faulty. This is because it will be hard for non-faulty code to suppress a fault within a method. Therefore, a non-faulty code snippet is not going to stop a method from being defective. For example, if an IF statement was coded incorrectly within a method, a non-faulty expression statement above or below the If statement is not going to cause that method to magically become non-faulty. It is easier to prove a significant association between a code snippet and faulty code than it is with non-faulty code.

8.1.3 RQ2: Do the associations between code snippets and faultiness change as a system evolves?

The association between code snippets and faultiness does change as the system evolves. Code snippets that were significantly associated with faults in one release of EJDT, did not appear in another and vice versa. As EJDT evolved over the three releases, the number of snippets significantly associated with faulty methods rose from 6.74% to 14.36%. This finding suggests that an increasing number of faults are added as a system ages and the code base increases. Lehman [1980] was one of the first to describe the existence of these evolutionary problems. One of Lehman [1980]'s laws of software evolution states that as a system evolves, unless it is properly maintained, its complexity will increase. This increase in complexity could cause an increase in the number of defects.

Despite the number of code snippets changing their association with faults over time, there are still 12% of code snippets that are significantly associated with faults across all three of the releases. This 12% could be used to help predict future defects within EJDT. This 12% is much higher than the 1% when AspectJ and ArgoUML are included. This result shows that using code snippets within a system, could be useful at predicting where defects may be in future releases. When the releases are close together, the releases have a greater number of code snippets significantly associated with faults. More than half of the unique code snippets significantly associated with faults in EJDT 3.0 are also significantly associated with faults in EJDT 3.1. Only a third of code snippets significantly associated with faults in EJDT 3.0 are also significantly associated with faults in 2.0. This suggests that as the system matures, faults are found and begin to not become repeated. This information could be used to develop a dynamic list of known code structure problems currently within a software system. This dynamic list could be similar to those proposed by Hassan and Holt [2005] and Kim et al. [2007]. Hassan and Holt [2005] used process metrics to create a list to show the top ten subsystems that are likely to have problems. This list dynamically updates to reflect future risks. Hassan and Holt [2005] created four different lists: most frequently modified, most recently modified, most frequently fixed and most recently fixed. These lists were able to show managers where potential defects were within a system. Kim et al. [2007] wanted to predict the most fault prone files using process metrics (e.g. how files have changed and coupled files that have changed). Kim et al. [2007] used two cache lists to hold the information of the most fault prone entity locations - BugCache and FixCache. BugCache held the most recent fault locations and FixCache held where a defect had been fixed. These caches were used to indicate potential sources of defects, with 80-85% recall and 73% to 95% precision at the file level. This top ten list could be replicated with code snippets and provide a short term defect prediction. The dynamic list would have the top code snippets known to currently be significantly associated with faults, and developers could see this list as they prepare to develop the next release of the system. The list could be sorted by defect density, effect size or any way the developers chose. When new code snippets are found to be significantly associated with faults as the system matures, they are added to the list, with old code snippets being removed. This will help the developers as they will know the current coding structures that are causing problems. The developers could be encouraged to discover why this coding structure is causing problems within the

system or to just stop using the coding structure. The dynamic list could help stop some defects from being introduced to the system and causing problems for the end users.

8.2 Other Findings

This section will discuss some of the other outcomes of the results that do not specifically answer the research questions.

8.2.1 Other uses for Cross-system Code Snippets Significantly Associated with Faults

The 201 code snippets do not have to be limited to just defect prediction models. The code snippets could be useful within software testing and during development. For example, the 201 code snippets could be used to create a list of coding constructs that developers/testers should be made aware of. This list could be known as a *'Code construct smells'* (CCS) list (Appendix A.2 has the full list of code snippets to be included in the CCS). The CCS could be used during the development of a system in two different ways. Firstly, the CCS could be used during the actual coding of a system. When developing a particular system, an IDE could make developers aware that they have just created a piece of code with a CCS within it. A developer could be extra vigilant when developing that piece of code or decide to use a different construct that has not been shown to cause defects. Sometimes choosing to use a different construct may not be available, however the extra care the developer may have taken may help reduce the chance of a defect occurring. This kind of awareness system is already implemented in some IDEs. The Netbeans[3] IDE makes users aware of mistakes in their code which will cause compilation errors and will warn users if code does not meet good coding practices (e.g. it will warn a user if they have not used brackets on an IF statement). For example, the code snippet `MEMBER_SELECT; IDENTIFIER; IDENTIFIER` is significantly associated with defects in all five systems. A developer uses this construct when coding a new method. The IDE that developer is using creates a warning, making the user aware that they have used a known defect-prone code construct. The developer can check their code and make sure it is correct, or decide to use an alternative coding construct. The warning system could eventually offer the user an alternative construct to use. This will require research into how faulty constructs change when they are fixed. Such work is beyond the scope of this dissertation but could constitute further work. Secondly, the CCS could be used during the static testing phase of development. Static testing or static code analysis is a software testing method which involves examination of the systems' code to determine if it meets specific requirements [Kaner et al. 2000]. One static testing technique is static code analysis. Static code analysis is a form of code review which analyses code without actually executing it [Kaner et al. 2000]. Static code analysis is generally used to find defects or ensure conformance to coding guidelines [Kaner et al. 2000]. The CCS could be used for static code analysis. It could help developers focus on parts of the system that have been shown to have these coding constructs. When developers/testers know where the potential problems exist, they could act upon them, much in the same way as they would during the real time development of the code. This work with the CCS list during real time development or as a static testing tool could help reduce the number of defects that are released into the system. The development stage is one of the cheapest stages to remove defects from within a system [Pressman 2001], so finding as many defects as possible at this stage will help save companies money.

[3] http://www.netbeans.com

8.2.2 Full-set v Sub-set of Code Features

The technique described in this dissertation gathers all possible code features within a chosen software system(s). The technique creates an unbiased exhaustive list of code snippets that have been used in the development of a software system. This list could be known as the full-set of code features that are available within a developed software system. Previous work creating defect prediction models has focused on using a small number of independent variables. Most of the traditional models have only used a sub-set of coding features. Cyclomatic complexity (CC) is one such code feature that has been used [Mende and Koschke 2010, Menzies et al. 2007, Ohlsson and Alberg 1996, Turhan et al. 2009]. CC uses only a small number of possible code features to determine if a module is defective or not. Studies have scrutinised the CC metrics in the past, claiming that CC uses arbitrary constants (in the error estimate and programming time calculations) and may only be a proxy for LOC [Fenton and Pfleeger 1998, Hamer and Frewin 1982, Shen et al. 1983, Shepperd and Ince 1994]. However, this dissertation indicates that there are code snippets significantly associated with faults that are linked to CC measures. The technique described has been able to find some problematic code snippets that relate to coding constructs which developers have been commonly encouraged to avoid. For example, the use of != relates to the known logical difficulty associated with correctly using double negatives. Similarly the use of && relates to the known difficulty associated with creating complex conditional statements. Such complex conditional statements are part of the calculation of CC as they increase the number of logical paths. The results in this dissertation provide some empirical evidence to substantiate the use of McCabe [1976]'s cyclomatic complexity in defect prediction. Some of the code snippets significantly associated with faults, could be used to help confirm the use of specific object oriented metrics. Coupling/message chains have been shown to have a negative effect on code correctness based on the results of this dissertation. Other OO metrics are mainly focused at the class level and so cannot to be found using the code snippet technique as code snippets are at a lower level of granularity. There is no evidence in the results to substantiate use of other software code metrics. For example, the significant association between a code snippet and faults can not substantiate the use of the LOC metric. The code snippets are too short to be used to confirm that LOC is a good or bad predictor of faults. The maximum code snippet length of seven will mean that the LOC involved in that snippet will be very short. Figure 1.1 shows a simple five line method and that five line method has a sequence length of 16. Code snippet length is likely just to be a proxy to LOC. Also, as shown in Figure 6.3 as a code snippet length increases, the chance of that code snippet being faulty decreases.

The dissertation results also show that there are many coding constructs that are fault-prone which may have not been studied before. The results may explain the current ceiling of defect predictors [Menzies et al. 2007]. Over-dependence on single metrics such as cyclomatic complexity or using the OO metrics in defect prediction will restrict predictive performance to only a sub-set of defects. There are many more coding structures that could be exploited that may produce better defect prediction results. A defect prediction technique that has used many different code features has been shown to have the best results in defect prediction [Hall et al. 2012]. Shivaji et al. [2009] used a modified bag of words (BOW+) approach to select the best features of code to use in a defect prediction model. Shivaji et al. [2009]'s results show that using combinations of features can be successful within defect prediction. The many features discovered in this dissertation could be used effectively within defect prediction. The full set of features could be used, or the best sub-set of features could be used. The information gain (gain ratio) approach used by Shivaji et al. [2009] could be used to determine the most effective sub-set of code snippets that could be used within a defect prediction model. A sub-set of code snippets may have to be used, as not all of the coding structures found may be effective at finding potential defects. The

effect sizes and defect densities of each of the code snippets significantly associated with defects could also be used to find the more useful code snippets. The effect sizes of the code snippets measures the practical significance of a statistically significant result. Effect sizes have been given for all reported code snippets within Chapter 6. The effect size will determine if the difference between the expected result and actual result is large enough to have practical value. Effect sizes have not been reported well in the past [Kampenes et al. 2007]. However, a few studies have reported effect sizes in studies involving the association between code bad smells and faults [Hall et al. 2014, Sjoberg et al. 2013, Yamashita and Counsell 2013]. Effect sizes are important in defect prediction models as effort in finding defects should be prioritised to coding constructs that have a large effect on faults. This work determining if there is an effective code snippet sub-set has not been undertaken as it was not the intention of the dissertation to find the code snippets that most effectively predicted defects. The aim of the dissertation was to determine if any code structures could be associated with faulty code. The investigation into the most effective code snippets would make for very interesting further work and could determine if there are a sub-set of significant code snippets that would break the Menzies et al. [2007] ceiling.

8.2.3 Smelly Code Snippets

A previously found 'code smell' has been found within some of the most prevalent code features that have been significantly associated with faults in this research. The snippets: `METHOD_INVOCATION; MEMBER_SELECT` and `MEMBER_SELECT; IDENTIFIER` are all significantly associated with faults. These snippets relate to the *message chain* code bad smell. The *message chain* code smell was identified as a problem by Fowler and Beck [1999] and has been reported to have a small but significant effect on faults Hall et al. [2014]. Low coupling is a desirable design principle in object oriented programming [Stevens et al. 1974] as low coupling helps maintain high readability and maintainability. A *message chain* will introduce coupling. High coupling makes the software harder to maintain as changes undertaken in one part of the chain, will impact on other parts of the chain. The other parts of the chain could become defective if they are not modified to reflect the change. The longer the chain, the more changes that are needed to be made and therefore the higher chance of a defect occurring. The results in this dissertation show that kinds associated with message chains could have a big effect on the chance of that method being defective. The findings of this dissertation add further evidence confirming the problem of *message chains* within code. There is no evidence within our results to suggest that any of the other code smells found by Fowler and Beck [1999] to be associated with faults using code snippets. This is due to the smells being at a higher granularity than what is being investigated in this dissertation.

The results in this dissertation show that the `IDENTIFIER` kind occurs in many of the frequently occurring snippets that are significantly associated with fault-prone code. The `IDENTIFIER` kind itself is significantly associated with faults in all three of the EJDT releases. Identifiers are frequently changed and could easily be programmed incorrectly. Research into lexicon bad smells has focused on the bad use of identifiers and the language of identifiers within code [Abebe et al. 2012, Arnaoudova et al. 2010, Butler et al. 2009]. Lexicon bad smells are where developers have used short terms (e.g abbreviation or acronym) or meaningless terms (e.g. `foo` and `bar`) as identifiers. Poor quality lexicon for identifiers has been shown to be associated with the introduction of faults [Butler et al. 2009]. Lexicon bad smells have been able to improve defect prediction when used alongside traditional source code metrics [Abebe et al. 2012]. Arnaoudova et al. [2010] investigated the link between identifiers and fault proneness and found that identifier terms with high entropy had a greater chance of being faulty. This work could help suggest why the `IDENTIFIER` term is shown to be significantly associated with faults. Identifier terms could be changed frequently during defect fixes within each of the EJDT releases. The results presented in this

dissertation could add further evidence to the work being undertaken by Abebe et al. [2012], Butler et al. [2009] and Arnaoudova et al. [2010], that there is a relationship between poor quality identifiers in code and defects.

8.2.4 Code Snippet Length - Does it Matter?

As a snippet length increases, the chance of a snippet being associated with a fault decreases. This result suggests that the length of the code snippet is important at determining whether it will be defective or not. It indicates that a larger Java construct has a smaller chance of being defective i.e. longer snippets lose their predictive power. A reason for this is due the construction of the code snippet. A code snippet is an ordered sequence of kinds. Each kind will have its own probability of becoming defective. Some kinds will have a higher predictive power than others. The kinds that have a lower predictive power could be suppressing the ability of that code snippet to predict potential defects. For example, the kind `VARIABLE` is not significantly associated with faults in EJDT 2.0. This could limit the predictive power of a code snippet that has a `VARIABLE` kind or multiple `VARIABLE` kinds within it. As the sequences get longer, there is a greater chance of a code snippet containing a greater number of kinds that could be limiting the defective power. For example, the coding construct of nested IFs could be seen to increase the number of paths taken by a program and therefore increase CC. This coding construct would be represented as a `IF; IF` code snippet. However, the `IF; IF` code snippet may be significantly associated with faults, but may never appear in analysis under the current method. This could be due to the number of kinds that will appear between the two `IF` kinds.

In order to solve the problem of insignificant kinds causing longer code snippets to lose their predictive power, the current restriction of using the ordered kinds as code snippets could be taken away. The code snippet method presented in this paper extracts all possible permutations of kinds from an ordered seqeunce that are one to seven kinds in length. An alternative method could be that instead of extracting permutations, all possible combinations of kinds are extracted from a sequence. For example, a method is sequenced in to the following kinds: `METHOD; MODIFIERS; INSTANCE_OF; CATCH; VARIABLE; EXPRESSION_STATEMENT; IDENTIFIER`. Using Equation 8.1 the sequence will have a maximum of 126 combinations of code snippets, instead of the original 28 permutations. These 126 combinations could be reduced by removing possible duplicate code snippets that are extracted from the sequence. The new combinations of the code snippets could discover new code snippets that could be used in defect prediction that were not found using the method described in this dissertation. These new combinations could include longer coding constructs that have been masked by other snippets that appear between them in the original sequences.

$$Combinations = \sum_{r=1}^{r=n} \frac{n!}{(n-r)!(r)!} \tag{8.1}$$

Where n = snippet length.

8.2.5 The Distribution of Code Snippets

The results have shown most of the code snippets that are significantly associated with faults appear in a small proportion of methods. In total 99% of all the code snippets found that are significantly associated

with faults appear in less than 10% of the methods across all five systems. Therefore, the code snippets that are in the 1% are important for defect prediction. The 1% are the most frequently occurring within all the systems. These 1% are the Java coding constructs that developers are most commonly programming incorrectly. This information is of interest to both developers and defect predictors. Developers could use the top 1% code snippets as a list to focus on both during development and during testing. Focusing on these most frequently occurring code snippets could potentially mean that they find previously latent defects within the software system.

The fact that most of the significant code snippets are located in a small number of methods shows that coding constructs that cause faults are a very localised problem. This finding coupled with the discovery that the developers of the five systems have used less than 1% of the possible coding constructs available may explain why Weimer et al. [2009]'s study was so successful. Weimer et al. [2009] used genetic programming to automatically repair program defects. Weimer et al. [2009]'s technique evolved the program variants until one was found that repaired the defect and also passed the required test cases. Weimer et al. [2009] suggest that their algorithm has a "potentially infinite-size" search space it must sample to find a correct repair. The findings in this dissertation suggest that this search space is much lower than Weimer et al. [2009] expects and may explain some of the success of the algorithm. The code snippets could also help the Weimer et al. [2009] algorithm by further reducing the search space they have to search through. If a code snippet is known to cause problems, the algorithm could ignore this coding construct in its search for a repair for a defect.

The discovery that less than 1% of the possible code snippets have been used in making the five systems helps add to the research that software code is predictable. Gabel and Su [2010] studied the uniqueness of source code. Gabel and Su [2010] were able to measure the uniqueness of over 6,000 different projects, covering 430 million LOC. Gabel and Su [2010] showed that the level of uniqueness of software for samples of code between one and seven lines in length was very low. This could be a possible reason as to why there are very few code snippets being used. The code snippets found in this dissertation are a maximum seven kinds in length and seven kinds is unlikely to cover more than seven LOC. Hindle et al. [2012] worked on the premise that although software code is artificial, it is the "natural product of human effort". Therefore, due to this human effort, software code will have the same properties as natural language and can be largely regular and predictable. Hindle et al. [2012] used lexical tokens from a variety of Java and C systems. These lexical tokens were used to compare the cross-entropy of the software code across the systems. The cross-entropy will measure how surprising a test document is to a model based on a corpus [Hindle et al. 2012]. Hindle et al. [2012] results showed that there was a high level of regularity in the source code analysed and that this regularity came from the naturalness of the source code. Hindle et al. [2012] used their results to create a successful code completion engine for the Eclipse IDE. The results in this dissertation helps add further evidence to the work by Gabel and Su [2010], Hindle et al. [2012]. Whilst their work focused on the lexical analysis of the source code, the results in this dissertation show syntactical evidence that few constructs are used in software code.

8.2.6 Code Cloning and Code Snippets

Some significant code snippets found in a version of EJDT 2.0 may have been due to code cloning within the system. These code clones were found due to a large number of code snippets appearing very regularly and always being defective. The code snippets were found mostly in getter and setter methods and had the same syntactical patterns. The code snippets found were restricted to one part of the EJDT 2.0 system and do not appear faulty in later releases. A manual inspection of the classes

that were involved in the cloning led to the conclusion that they were part of a visitor pattern that had become faulty. The defect along with the coding cloning meant that many different methods had to be changed within EJDT 2.0. This suggests that sometimes code cloning may not be good practice. If a cloned piece of code does become faulty, then mass fixes may have to take place. It is not always clear where these changes may need to take place [Roy et al. 2009]. The finding that a particular problematic section of code seemed to be related to the use of code cloning resonates with previous studies reporting code clones as fault-prone [Juergens et al. 2009]. The finding also shows that code snippets could be used to find code clones within a chosen system. The use of the AST has been successful in the past at finding code clones [Baxter et al. 1998, Raza et al. 2006, Wahler et al. 2004, Yang 1991], but the use of individual kinds and code snippets to find clones has not been investigated.

8.2.7 Uses for Code Snippets

In summary this dissertation has highlighted several uses for the code snippets. These uses range from normal software prediction to using code snippets to help automatic patching algorithms. The code snippets could be used in defect prediction. Different snippets could be used to predict defects in different circumstances. If cross-project defect prediction was to be carried out, the 201 code snippets that are significantly associated with faults across all five of the systems analysed in this dissertation could be used. The code snippets that were significantly associated with faults in one release of a system, could be used to predict defects in a later release. Also, a dynamic list of current code snippets significantly associated with faults could be created to help predict defects in the short term for a selected system. In each of these techniques, the code snippets could be refined further to discover a sub-set of code snippets that create more accurate predictions. The same lists or sub-lists could also be used during development. During the development stage the code snippets significantly associated with faults could be used to create a warning system for developers. This warning system could suggest to the developer potential code structure problems that have been shown to be problems in the past. The developers could then change the code that they have just created, or choose to make sure it is performing correctly. The code snippets significantly associated with faults could also be used away from defect prediction. The code snippets could be used to examine how a program changes over time. The evolution of the code snippets could be used to indicate where certain coding constructs are mostly used and if any coding constructs change over time. The results in this dissertation have shown that as a system matures, there are more coding structures developing faults. The results also show that these coding structures change over time as not all the same code snippets are significantly associated with faults in all three releases of EJDT. The code snippets could be used to examine other code evolution phenomena. The code snippets could also be used to help improve the automatic patching of systems. The results of the dissertation show that there are very few coding constructs being used widely within the software system. The code snippets significantly associated with faults are located in a very small search space. These code snippets could be used to help further enhance the automatic patching algorithm developed by Weimer et al. [2009]. The algorithm could be improved as the snippets could help reduce the search space of the algorithm further by indicating known faulty paths. These faulty paths could be ignored when the algorithm is searching for a potential patch.

The code snippets themselves, not just the code snippets significantly associated with faults could be used within code cloning research. The code snippets could indicate potential code clones within a software system. During the examination of the code snippets in this dissertation, potential code clones were found due to code snippets appearing many times and appearing always faulty. The AST has already been used within code cloning research, however not at such a fine grained level. The code

snippets could help indicate smaller code clones within a software system, or help improve the discovery of type III code clones.

CHAPTER 9

Conclusion

"Don't worry if it doesn't work right. If everything did, you'd be out of a job."

– Mosher's Law of Software Engineering

This chapter concludes the dissertation. Section 9.1 concludes the overall findings of the research undertaken during the course of the PhD. Finally, Section 9.2 outlines the contributions to knowledge that have been discovered during the course of this dissertation.

9.1 Overall Findings

This research has concentrated on empirically investigating the association between Java code snippets and the faultiness of the Java methods that they are located within. The background research of this dissertation shows that there has been plenty of research that has tried to use many different code metrics and/or process metrics to predict where defects may be in a software system. However, there is evidence that the current methods are not fine grained enough and have hit a predictive ceiling. Research has been conducted into finer grained methods to predict defects, but these methods have focused on using the text of the code and not the actual constructs that are available via the AST. This dissertation presents a new, finer grained technique to examine the Java code for use in defect prediction.

The background research presented in this dissertation highlights that many defect prediction metrics rely on the use of only one or two programming constructs (e.g. branching points in cyclomatic complexity). Research has shown that techniques that combine various metrics perform better defect prediction. The technique presented in this dissertation does not restrict itself to just one or two programming constructs, the technique can isolate all programming constructs that have been used in a particular system. This could help defect prediction models in the future as it could highlight all the programming constructs that are the most vulnerable to faults. Having a bigger corpus of programming constructs that are vulnerable to faults could aid defect prediction models in making better decisions or to the discovery of defects that previous techniques could not find. Finding one or two defects that would not have been found with the use of previous techniques could help save a lot of money for software development companies.

The main aim of this research was to determine if there are associations between Java code snippets and faulty code. The work conducted in this dissertation has shown that the technique presented has been able to significantly associate specific Java code snippets and faults. There are thousands of code snippets that are significantly associated with faults within all five of the systems studied. Some of the code snippets that are significantly associated with faults indicate that research performed in the past has been correct. For example, message chains is one of the programming constructs that have been shown to appear frequently in the code snippets significantly associated with faults. Message chains have been

an indication in the past of a programming bad 'smell' and can increase cyclomatic complexity. The results presented in this dissertation may help provide further empirical evidence for these previously reported coding problems.

The research does not find any significant associations with non-faulty code. There are no Java programming constructs that are considered to be safe. This is not to say there are not any Java constructs that are always non-faulty, just there is no evidence to say that there are code snippets that are significantly associated with non-defective code. This makes sense as a programmer could make a mistake with any construct that they are implementing, also the presence of a non-defective piece of code may not suppress the chance of a method becoming faulty. This result implies that developers should work on defect avoidance, rather than using constructs that have been shown to be associated with non-faulty code.

There are only a few code snippets that are significantly associated with faults across releases in the EJDT system. As the system matures, more code snippets become significantly associated with faults. This finding means that some code snippets could be used to discover faults in future releases. More of the same code snippets are significantly associated with faults in closer releases, so code snippets could become less useful as time goes on. This is an important finding as it shows that as a software system matures, the software faults that were in a previous release have been fixed and are not being repeated. This result however does impact on the usefulness of a predictor based on older releases in predicting future defects. Code snippets could be used from one release to the next to advise developers of coding constructs that were significantly associated with faults. The developers could use this information to possibly reduce the amount of defects in the next release.

To summarise, the results gathered in this dissertation have found an answer to each of the research questions posed. To answer RQ1a, there are code snippets that are significantly associated with faulty code. The answer to RQ1b is that there are no code snippets that have been identified as being significantly associated with non-faulty code. Finally, the answer to RQ2 is that the association between code snippets and their significant relationship with faulty code does change as a system evolves.

9.2 Contributions to Knowledge

This dissertation has contributed knowledge to the software engineering community in a number of ways, both in the direct results of the research questions and in the methodology that has reached those answers.

The dissertation has developed a novel technique that can sequence Java code. This sequencing technique is relatively simple to apply to any system with a code repository and defect tracking system. These sequences are made up of all the Java constructs within a method. This technique has allowed for the extraction of Java code snippets, which allowed code snippets significantly associated with faulty code to be identified. This technique is not limited to be used in Java, it could be modified to work with any object oriented language which has an AST available. The sequences are not limited to just methods, the sequences could be as large as a class, or as little as an identifier. The sequencing of code has uses within the wider software engineering community and should not be restricted to defect prediction alone.

The use of the sequence technique has led to the introduction of the code snippet. A code snippet is a novel representation of Java code constructs This dissertation has successfully associated some of the discovered code snippets with faults. These code snippets that have been significantly associated with

faults could be used to develop new defect prediction models. They could prove successful as the code snippet idea does not bias itself on one, or a few constructs in code that could be defective, like many other techniques within software engineering. The code snippet technique is also the first to use the AST for this purpose, other finer grained techniques have focused on using the text of the code.

During the development of the technique for the dissertation, the SZZ algorithm described by Śliwerski et al. [2005], has been made into a functioning program in Java code. This implementation can be used by other developers to find possible defective code in different systems other than those that have been used in this dissertation. The technique that has been developed has high recall and precision and found more bug links that that of the original SZZ authors. This research has allowed the development of a bug-link database and relating defect data database. These two databases could be used by fellow researchers to replicate this study or perform a wide range of new analysis. The defect data database contains all the defects in each of the systems at line level. This work has also led to the creation of Git repositories of two of the four CVS repositories that Zimmermann[1] used to undergo his analysis. These Git repositories are also made available for wider use in the research community.

The work in this dissertation could be used within many research communities. Within defect prediction the results could be used to create new finer grained defect prediction models. Current models that use a variety of different code features have been shown to have the best defect prediction performance. The code snippet models could be based on sub-sets of the code snippets significantly associated with faults. These sub-sets could just be the code snippets that have been shown to be significantly associated with faults across all five systems, or a sub-set based on the information gain of each of the snippets. Code snippets could be used be used to warn developers of coding constructs that have been a problem in the past. These warnings could help the developers decide to either use a different construct, or take extra care in developing that construct. They could also be used to perform static code analysis to discover areas of a system that have code structures significantly associated with faults. Effort could then be focused on these areas of the system to eliminate any potential defects. This could all make a big impact on the cost to companies. Money will be saved if code snippets identify defects earlier during the development of a system, rather than said defects manifesting themselves as faults later on in the systems lifeline.

[1] http://msr.uwaterloo.ca/msr2008/challenge/

Chapter 10

Future Work

"There is always one more bug."

– Lubarsky's Law of Cybernetic Entomology

There is a lot of potential future work that could be undertaken based on either the technique or the results of the technique used in this dissertation. This potential future work is not presented as an exhaustive list, but shows how significant the results of this paper could be to the research community.

10.1 Using the Code Snippets for Defect Prediction

The code snippets presented in this dissertation or other code snippets significantly associated with faults found in other systems could be used to predict defects. The code snippets could be used to predict defects in future releases of the current system, or used to predict defects in a totally different system. The code snippets that have been found to be significantly associated with faults in this dissertation could be used as a starting point in this analysis or new code snippets could be found based on other systems. These sets of code snippets significantly associated with defects could be further refined to find the most effective sub-set. Finding the most effective set of code snippets to perform defect prediction could be done by using information gain analysis. The full or sub-set of defects could create a defect prediction model that could break the current performance ceiling.

10.2 Using the Code Snippets for Defect Prevention

A defect warning system could be implemented using the code snippets found within this dissertation. This defect warning system could take the form of an IDE plugin. The plugin could use the full or modified sub-sets of code snippets significantly associated with faults. The plugin would warn programmers during development about certain coding structures that have been associated with faults. If the developer has used a code construct that has been significantly associated with faults, the warning system will alert the developer. The developer could then make a decision to use a different coding structure, or cause the developer to focus more on the particular construct so a defect is not inserted. The plugin could rely on the help of a static or a dynamic code snippet list. A dynamic list would change as new code snippets significantly associated with faults are discovered during the development phase. This list will update as development continues with the most popular/most detrimental coding constructs. The dynamic list concept is based on the work by Hassan and Holt [2005] and Kim et al. [2007]. This plugin could stop a number of defects before the testing phase. Eliminating defects this early will help save companies money.

10.3 Code Snippet Changes

The technique used in this dissertation has a maximum limit of seven kinds for each possible code snippet. This limit is in place due to computational constraints. It should be investigated if there are longer code snippets that also could be significantly defective. There could be possible longer code snippets available that point to faulty code and ones that could be used in defect prediction. Also, there could be a length limit to possible significant defective code snippets.

Further work could focus on reducing the amount of kinds available. Reducing the amount of kinds could help discover further code constructs that are significantly associated with faults. These code constructs may not have been discovered using the technique introduced in this dissertation as there are kinds that are masking the relationship. Work could be undertaken to ignore certain kinds in the AST as some kinds may not have an impact on the defectiveness of a coding construct. This will allow the technique to focus on kinds which make the most impact on software faults. This work could induce combining kinds so that they count as one. For example, the `BLOCK` kind could be combined with the `METHOD` kind as they frequently appear together, so the search space is reduced. These combination would need to be investigated further.

The restriction that the code snippets have to be in order of the method sequence could be lifted. This could help discover bigger constructs that are significantly associated with defective code, without having to increase the length limit of the code snippet. For example, the bigger constructs that could be found could be nested `IF` statements or switch statements with many cases. This work into finding further coding constructs that are significantly associated with faults, could help increase the effectiveness of the defect prediction or prevention models. This increase in effectiveness could lead to more money being saved by companies that apply these models.

10.4 Code Snippets and the Programmer

An investigation into if the programmer has any impact on the Code snippets and the level of defectiveness. It could be investigated if an inexperienced programmer develops different code snippets to an experienced developer. Also, work could be undertaken to determine if code snippets developed by each of the different levels of experience have a greater or less chance of becoming defective. This could also be extended to the time when the code was written (i.e. do code constructs developed on Fridays introduce more defects?) This work could show the impact of an individuals work pattern and could help show if some code constructs are too difficult for some programmers to understand and implement correctly.

10.5 Evolution of Code Snippets

This dissertation involved tracking code snippets over three different revisions of one system. It could be investigated if certain code snippets are significantly associated with faults at different points in time within a system. Analysis could be undertaken to determine if there are certain code snippets that are prevalent at the beginning of a new system, or certain code snippets that occur at a major release. This information could help develop better models for defect prediction. This information could also help defect prevention. Developers could be made more aware about code snippets that are prevalent at a

certain milestone in a system, so these coding constructs are better managed at these times. This could reduce the amount of defects released and therefore could save the companies time and money fixing these defects at a later date.

10.6 Patching/Code Completion using Code Snippets

Code snippets could be used to automatically repair defective code. There is a lot of change information available when defects are fixed. Using this change data it could be investigated how a defective code snippet is changed when it is fixed. This coupled with the fact that there are only very few possible coding constructs used (less than 1%) could help develop an effective automatic patching algorithm. Work could be completed to determine if there are non-defective 'cousins' of code snippets that are significantly associated with faults. These cousins are code snippets that have been used to fix a defective code snippet. This work will provide a set of potential fixes for certain problems. These fixes could be used to create the automatic patching algorithms. The fixes could also be used by programmers during the development of code. If a warning system indicates that a coding construct has been used that is significantly associated with faults, this warning system could also offer potential alternatives for this particular construct. Also, when a defect has been discovered in a system, the fixes could provide different remedies for the particular defect. Work will need to be completed to see if these fixes actually exist, and then if they do, how effective in the fixing or prevention of defects.

10.7 Code Cloning

A potential code cloning problem has been found during the analysis of the code snippets. The technique highlighted several code snippets that appear very often and were always defective. These code snippets were part of a visitor pattern that had become defective. This visitor pattern relied on small cloned methods. Further work could be undertaken to determine if the code snippets could be used to discover other clones within a system. Code cloning research has used ASTs in the past, but not at the code snippet granularity. This dissertation was able to find a possible Type III code clone. It could be investigated if code snippets could find other code clones in other systems and different types of clones.

References

S.L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y. Gueheneuc. Can lexicon bad smells improve fault prediction? In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 235–244, Oct 2012. doi: 10.1109/WCRE.2012.33. (Cited on pages 2, 23, 95 and 96.)

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles. *Techniques, and Tools*, 11 (0011):00, 1986. (Cited on page 21.)

Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971. (Cited on page 14.)

John Aldrich. Correlations genuine and spurious in pearson and yule. *Statistical Science*, pages 364–376, 1995. (Cited on page 41.)

Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635901. URL http://doi.acm.org/10.1145/2635868.2635901. (Cited on page 66.)

C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *Software Engineering, IEEE Transactions on*, 33(5):273–286, May 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.1005. (Cited on page 14.)

Erik Arisholm and Lionel C Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17. ACM, 2006. URL http://dl.acm.org/citation.cfm?id=1159738. (Cited on page 19.)

Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83 (1):2–17, 2010. (Cited on page 1.)

Venera Arnaoudova, Laleh Mousavi Eshkevari, Rocco Oliveto, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *Proceedings of the International Conference on Software Maintenance (ICSM) - ERA Track*, pages 1–5, 2010. (Cited on pages 23, 95 and 96.)

AssociatedPress. Toyota "unintended acceleration" has killed 89 - cbs news. Webpage, 2010. URL http://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/. Accessed 07th October 2014. (Cited on page 9.)

Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 97–106, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882308. URL http://doi.acm.org/10.1145/1882291.1882308. (Cited on page 48.)

Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014), Hong Kong*, 2014. (Cited on page 66.)

Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996. (Cited on pages 18 and 19.)

I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, 1998. (Cited on pages 35 and 98.)

BBC. BBC News - Samsung pull Galaxy S3 update after complaints". Webpage, 2013. URL http://www.bbc.co.uk/news/technology-24995544. Accessed 25th November 2013. (Cited on page 1.)

Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Looking for bugs in all the right places. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 61–72, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1. doi: 10.1145/1146238.1146246. URL http://doi.acm.org/10.1145/1146238.1146246. (Cited on pages 1, 14, 20 and 90.)

Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 11–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-722-3. doi: 10.1145/1294948.1294953. URL http://doi.acm.org/10.1145/1294948.1294953. (Cited on page 20.)

David Binkley, Henry Feild, Dawn Lawrie, and Maurizio Pighin. Software fault prediction using language processing. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 99–110. IEEE, 2007. (Cited on page 23.)

David Binkley, Henry Feild, Dawn Lawrie, and Maurizio Pighin. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 82(11):1793–1803, 2009. (Cited on pages 2 and 23.)

Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009a. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595716. URL http://doi.acm.org/10.1145/1595696.1595716. (Cited on pages 7, 43 and 48.)

Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 109–119. IEEE, 2009b. (Cited on pages 1 and 48.)

Forenza Brady. Cambridge university study states software bugs cost economy $312 billion per year. Webpage, 2013. URL http://www.prweb.com/releases/2013/1/prweb10298185.htm. Accessed 24th October 2013. (Cited on pages 1 and 9.)

Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91–121, 1999. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=748920. (Cited on page 19.)

Kate Brinton and Kim-An Lieberman. What is DNA Fingerprinting? http://protist.biology.washington.edu/fingerprint/whatis.html, May 1994. Accessed May 2011. (Cited on page 26.)

S. Butler, M. Wermelinger, Yijun Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 31–35, Oct 2009. doi: 10.1109/WCRE.2009.50. (Cited on pages 23, 95 and 96.)

Jean Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254, 1996. (Cited on page 50.)

V.U.B. Challagulla, F.B. Bastani, I-Ling Yen, and R.A. Paul. Empirical assessment of machine learning based software defect prediction techniques. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 263 – 270, feb. 2005. doi: 10.1109/WORDS.2005.32. (Cited on page 49.)

Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=295895. (Cited on page 17.)

S.R. Chidamber, D.P. Darcy, and C.F. Kemerer. Managerial use of metrics for object-oriented software: an exploratory analysis. *Software Engineering, IEEE Transactions on*, 24(8):629–639, Aug 1998. ISSN 0098-5589. doi: 10.1109/32.707698. (Cited on page 19.)

Collabnet. argouml.tigris.org. http://www.argouml.tigris.org, 2014. Accessed 28th July 2014. (Cited on page 43.)

CollinsEnglishDictionary. Collins English Dictionary. Book. (Cited on page 9.)

Tracy Connor. Obamacare's rocky start: What you need to know. http://usnews.nbcnews.com/_news/2013/10/22/21064709-obamacares-rocky-start-what-you-need-to-know, 2013. Accessed 24th October 2013. (Cited on pages 1 and 10.)

John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2003. (Cited on pages 32 and 33.)

Ana Erika Camargo Cruz and Koichiro Ochimizu. Towards logistic regression models for predicting fault-prone code across software projects. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 460–463. IEEE, 2009. (Cited on page 1.)

Tesse Und Daan. Empirical cycle, 2014. URL http://en.wikipedia.org/wiki/Empirical_research#mediaviewer/File:Empirical_Cycle.svg. Accessed 28th July 2014. (Cited on pages ix and 32.)

Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010. (Cited on pages xiii, 1, 10 and 18.)

Carrie Dann. Obama: 'The Affordable Care Act is not just a website' - NBCNews.com. http://www.nbcnews.com/news/other/obama-affordable-care-act-not-just-website-f8C11431491, 2013. Accessed 24th October 2013. (Cited on page 10.)

Adrianus Dingeman de Groot and JAA Spiekerman. *Methodology: Foundations of Inference and Research in the Behavioral Sciences*. Mouton, 1969. (Cited on pages ix, 31, 32 and 34.)

Norman K Denzin and Yvonna S Lincoln. *The SAGE handbook of qualitative research*. Sage, 2011. (Cited on page 32.)

Michael Dunn. Toyota's killer firmware: Bad design and its consequences. http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences, 2013. Accessed 07th October 2014. (Cited on page 9.)

Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008. (Cited on pages 32 and 33.)

Khaled El Emam, Walcelio Melo, and Javam C Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001. (Cited on page 19.)

Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, August 2000. ISSN 0098-5589. doi: 10.1109/32.879815. URL http://dx.doi.org/10.1109/32.879815. (Cited on pages 1 and 14.)

Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998. ISBN 0534954251. (Cited on pages 16 and 94.)

M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23 – 32, sept. 2003a. doi: 10.1109/ICSM.2003.1235403. (Cited on pages 37 and 46.)

Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 23–, Washington, DC, USA, 2003b. IEEE Computer Society. ISBN 0-7695-1905-9. URL http://dl.acm.org/citation.cfm?id=942800.943568. (Cited on page 46.)

Martin Fowler. Codesmell. http://martinfowler.com/bliki/CodeSmell.html, February 2006. Accessed 9 June 2014. (Cited on page 28.)

Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2. (Cited on pages 28, 29 and 95.)

FoxNews. Contractor says ObamaCare website fix will cost 121M. Webpage, 2014. Accessed 30th April 2014. (Cited on page 10.)

Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010. (Cited on page 97.)

Roger J Gagnon. Empirical research: The burdens and the benefits. *Interfaces*, 12(4):98–102, 1982. (Cited on page 40.)

Tihana Galinac Grbac, Per Runeson, and Darko Huljenić. A second replicated quantitative analysis of fault distributions in complex software systems. *Software Engineering, IEEE Transactions on*, 39(4): 462–476, April 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.46. (Cited on page 14.)

G. David Garson. *Binomial Test*, pages 69–70. SAGE Publications, Inc., 0 edition, 2004. doi: http://dx.doi.org/10.4135/9781412950589. URL http://dx.doi.org/10.4135/9781412950589. (Cited on pages xiii and 38.)

Lisa M Given. *The Sage encyclopedia of qualitative research methods*. Sage Publications, 2008. (Cited on page 33.)

Nils Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 311–320, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: http://doi.acm.org/10.1145/1985793.1985836. URL http://doi.acm.org/10.1145/1985793.1985836. 196. (Cited on page 29.)

T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. 26(7):653–661, 2000. doi: 10.1109/32.859533. (Cited on pages 1, 19 and 20.)

David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. The misuse of the NASA metrics data program data sets for automated software defect prediction. In *Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 96 –103, april 2011. doi: 10.1049/ic.2011.0012. (Cited on pages 1, 2, 22 and 23.)

T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, Oct 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.112. LOC. (Cited on page 14.)

T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. Developing fault-prediction models: What the research can show industry. *IEEE Software*, 28(6):96–99, 2011. doi: 10.1109/MS.2011.138. (Cited on page 42.)

T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6): 1276–1304, 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.103. (Cited on pages 1, 12, 14, 19, 22 and 94.)

Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2014. (Cited on pages 29 and 95.)

Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN 0444002057. (Cited on pages xiii, 15 and 16.)

Peter G. Hamer and Gillian D. Frewin. M.H. Halstead's software science - a critical examination. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, pages 197–206, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press. URL http://dl.acm.org/citation.cfm?id=800254.807762. (Cited on pages 16 and 94.)

A.E. Hassan and R.C. Holt. The top ten list: dynamic fault prediction. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 263–272, Sept 2005. doi: 10.1109/ICSM.2005.91. (Cited on pages 1, 20, 21, 22, 92 and 105.)

Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009. (Cited on pages 1 and 21.)

B Henderson-Sellers and A Henderson-Sellers. Sensitivity evaluation of environmental models using fractional factorial experimentation. *Ecological Modelling*, 86(2):291–295, 1996. URL http://www.sciencedirect.com/science/article/pii/0304380095000666. (Cited on page 19.)

Kim Herzig and Andreas Zeller. *Mining Your Own Evidence*, chapter 27. O'Reilly Media, Inc., October 2010. ISBN 9780596808327. (Cited on page 45.)

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012. (Cited on pages 65 and 97.)

EDS IBM et al. Abstract syntax tree metamodel (ASTM). *OMG Document*, 2013. (Cited on page 27.)

IEEE. IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010. doi: 10.1109/IEEESTD.2010.5399061. (Cited on pages ix, 1, 9, 10 and 12.)

IEEEReliability. IEEE Xplore: Reliability, IEEE Transactions on. Webpage, 2015. URL http://ieeexplore.ieee.org/xpl/aboutJournal.jsp?punumber=24. Accessed 19th November 2014. (Cited on page 7.)

Timea Illes-Seifert and Barbara Paech. Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs. *Information & Software Technology*, 52(5):539–558, 2010. (Cited on page 21.)

National Human Genome Research Insitute. Genome.gov | Deoxyribonucleic Acid (DNA) Fact Sheet. http://www.genome.gov/25520880, March 2011. Accessed 24th May 2011. (Cited on pages 2 and 26.)

National Human Genome Research Institute. Talking glossary: "chromosome". Webpage, 2014. URL http://www.genome.gov/glossary/index.cfm?id=33. (Cited on pages 2 and 25.)

Andrea Janes, Marco Scotto, Witold Pedrycz, Barbara Russo, Milorad Stefanovic, and Giancarlo Succi. Identification of defect-prone classes in telecommunication software systems using design metrics. *Inf. Sci.*, 176(24):3711–3734, December 2006. ISSN 0020-0255. doi: 10.1016/j.ins.2005.12.002. URL http://dx.doi.org/10.1016/j.ins.2005.12.002. (Cited on page 19.)

Alec J. Jeffreys, Victoria Wilson, and Swee Lay Thein. Hypervariable 'minisatellite' regions in human DNA. *Nature*, 314:67–73, 1985. (Cited on pages ix, 2, 25 and 26.)

Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003. (Cited on pages 26 and 27.)

Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: http://dx.doi.org/10.1109/ICSE.2009.5070547. URL http://dx.doi.org/10.1109/ICSE.2009.5070547. 164. (Cited on pages 29 and 98.)

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28:654–670, July 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019480. URL http://dl.acm.org/citation.cfm?id=636188.636191. (Cited on page 35.)

Vigdis By Kampenes, Tore Dybå, Jo E Hannay, and Dag IK Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11):1073–1086, 2007. (Cited on page 95.)

Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000. (Cited on page 93.)

Manju Karthikeyan. Human genetic screening, 1999. URL http://www.ndsu.edu/pubweb/~mcclean/plsc431/students99/karthikeyan.htm. Accessed 6 June 2014. (Cited on page 26.)

T. M. Khoshgoftaar, K Gao, and N. Seliya. Attribute selection and imbalanced data: Problems in software defect prediction. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 137–144. accept, 2010. (Cited on page 1.)

Taghi M. Khoshgoftaar and Naeem Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Softw. Engg.*, 9(3):229–257, 9 2004. ISSN 1382-3256. doi: 10.1023/B:EMSE.0000027781.18360.9b. URL http://dx.doi.org/10.1023/B:EMSE.0000027781.18360.9b. (Cited on page 19.)

Taghi M Khoshgoftaar, Nishith Goel, Edward B Allen, and Kalai S Kalaichelvan. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, 1996. URL http://www.computer.org/csdl/mags/so/1996/01/s1065.pdf. (Cited on pages 1 and 21.)

Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: 10.1109/ASE.2006.23. URL http://dx.doi.org/10.1109/ASE.2006.23. (Cited on pages 37, 46 and 48.)

Sunghun Kim, T. Zimmermann, E.J. Whitehead, and A. Zeller. Predicting faults from cached history. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 489 –498, may 2007. doi: 10.1109/ICSE.2007.66. (Cited on pages 1, 21, 22, 37, 46, 92 and 105.)

Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481 –490, may 2011. doi: 10.1145/1985793.1985859. (Cited on pages 23, 37, 49 and 55.)

B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *Software Engineering, IEEE Transactions on*, 21(12):929 –944, dec 1995. ISSN 0098-5589. doi: 10.1109/32.489070. (Cited on page 32.)

Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, 2002. (Cited on pages 36, 40 and 41.)

Barbara A Kitchenham, Emilia Mendes, and Guilherme H Travassos. Cross versus within-company cost estimation studies: A systematic review. *Software Engineering, IEEE Transactions on*, 33(5): 316–329, 2007. (Cited on page 90.)

Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*, 2007. (Cited on page 77.)

J. Krinke. Is cloned code more stable than non-cloned code? In *SCAM*, pages 57 –66, 2008. doi: 10.1109/SCAM.2008.14. (Cited on page 29.)

Lucas Layman, Nachiappan Nagappan, Sam Guckenheimer, Jeff Beehler, and Andrew Begel. Mining software effort data: preliminary analysis of visual studio team system data. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 43–46. ACM, 2008. (Cited on pages xiii, 21 and 22.)

Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68 (9):1060–1076, 1980. (Cited on page 92.)

S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485 –496, july-aug. 2008. ISSN 0098-5589. doi: 10.1109/TSE.2008.35. (Cited on page 49.)

Meredith Levinson. Lets stop wasting $78 billion per year. *CIO Magazine*, 15, 2001. (Cited on page 9.)

Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993. (Cited on page 19.)

Johannes Lyklema. *Fundamentals of interface and colloid science: soft colloids*, volume 5. Academic press, 2005. (Cited on page 26.)

Niccolò Machiavelli. *The Prince*. Yale University Press, 1997. (Cited on page 10.)

A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *Software Engineering, IEEE Transactions on*, 34(2):287–300, March 2008. ISSN 0098-5589. doi: 10.1109/TSE.2007.70768. (Cited on page 23.)

Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976. (Cited on pages 15, 19, 65 and 94.)

SM McGuigan. The use of statistics in the British Journal of Psychiatry. *The British Journal of Psychiatry*, 167(5):683–688, 1995. (Cited on page 40.)

T. Mende and R. Koschke. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 107–116, 2010. (Cited on pages 1 and 94.)

Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, page 7. ACM, 2009. (Cited on page 1.)

T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2 –13, jan. 2007. (Cited on pages 1, 16, 94 and 95.)

Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, December 2010. ISSN 0928-8910. doi: 10.1007/s10515-010-0069-5. URL http://dx.doi.org/10.1007/s10515-010-0069-5. (Cited on pages 1 and 22.)

Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 343–351. IEEE Computer Society, 2011. (Cited on page 90.)

Osamu Mizuno, Shiro Ikami, Shuya Nakaichi, and Tohru Kikuno. Spam filter based approach for finding fault-prone software modules. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 4–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: 10.1109/MSR.2007.29. URL http://dx.doi.org/10.1109/MSR.2007.29. (Cited on pages 2 and 23.)

Daniel Moody. Empirical research methods. http://www.itu.dk/~oladjones/semester3/advanceditmgtandsoftwareengineering/project/materials/whatisempiricalresearch1.pdf, 2014. (Cited on page 31.)

R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 181–190. accept, 2008. (Cited on pages xiii, 1, 19 and 20.)

MSR. Home - MSR 2014, 2014. URL http://2014.msrconf.org/. Accessed 31 August 2014. (Cited on page 45.)

N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. accept, 2010. (Cited on pages 1 and 21.)

Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005. (Cited on pages 1 and 21.)

Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006. (Cited on pages 90 and 91.)

NationalInstituteofHealth. Genetic disorders: Medicine plus. http://www.nlm.nih.gov/medlineplus/geneticdisorders.html, May 2014. Accessed 6 June 2014. (Cited on page 24.)

Allen P Nikora and John C Munson. Developing fault predictors for evolving software systems. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 338–350. IEEE, 2003. (Cited on page 21.)

Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *Software Engineering, IEEE Transactions on*, 22(12):886–894, 1996. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=553637. (Cited on pages 16 and 94.)

Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355, 2005. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1435354. (Cited on pages 1, 14, 20 and 90.)

David Patterson. Molecular genetic analysis of down syndrome. *Human genetics*, 126(1):195–214, 2009. (Cited on page 26.)

Dewayne E Perry, Adam A Porter, and Lawrence G Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 345–355. ACM, 2000. (Cited on pages 40 and 41.)

Jean Petric and Tihana Galinac Grbac. Software structure evolution and relation to system defectiveness. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*, 2014. (Cited on pages ix, 23 and 24.)

Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0072496681. (Cited on pages ix, 1, 9, 10, 11, 17 and 93.)

Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *Software Engineering, IEEE Transactions on*, 31(6):511–526, 2005. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1463233. (Cited on pages 1 and 21.)

F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR)*, pages 72–81, 2010. doi: 10.1109/MSR.2010.5463343. (Cited on pages 29 and 49.)

J. Ratzinger, H. Gall, and M. Pinzger. Quality assessment based on attribute series of software evolution. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 80–89, Oct 2007. doi: 10.1109/WCRE.2007.39. (Cited on pages 20 and 21.)

Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999. (Cited on page 90.)

Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – a tool suite for program analysis and reverse engineering. In *In Reliable Software Technologies, Ada Europe 2006 (LNCS 4006*, page 71, 2006. (Cited on pages 35 and 98.)

Marco Ronchetti, Giancarlo Succi, Witold Pedrycz, and Barbara Russo. Early estimation of software size in object-oriented environments a case study in a CMM level 3 software firm. *Inf. Sci.*, 176(5): 475–489, March 2006. ISSN 0020-0255. doi: 10.1016/j.ins.2004.08.012. URL http://dx.doi.org/10.1016/j.ins.2004.08.012. (Cited on page 19.)

J. Rosenberg. Some misconceptions about lines of code. In *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, pages 137–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8093-8. URL http://dl.acm.org/citation.cfm?id=823454.823905. (Cited on page 14.)

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, May 2009. ISSN 0167-6423. doi: 10.1016/j.scico.2009.02.007. URL http://dl.acm.org/citation.cfm?id=1530898.1531101. (Cited on pages 35 and 98.)

Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *School of Computing TR, Queen's University*, 115, 2007. (Cited on page 35.)

Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 18–27, New York, NY, USA, 2006. ACM. ISBN 1-59593-218-6. doi: 10.1145/1159733.1159739. URL http://doi.acm.org/10.1145/1159733.1159739. (Cited on page 21.)

Sam Scott and Stan Matwin. Feature engineering for text classification. In *ICML*, volume 99, pages 379–388, 1999. (Cited on page 23.)

G.M.K. Selim, L. Barbour, Weiyi Shang, B. Adams, A.E. Hassan, and Ying Zou. Studying the impact of clones on software defects. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 13 –21, oct. 2010. doi: 10.1109/WCRE.2010.11. 396. (Cited on page 29.)

Vincent Yun Shen, Samuel D. Conte, and Hubert E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. *Software Engineering, IEEE Transactions on*, (2): 155–165, 1983. (Cited on pages 16 and 94.)

M Shepperd and Darrel C Ince. A critique of three metrics. *Journal of Systems and Software*, 26(3): 197–210, 1994. (Cited on pages 16 and 94.)

Thomas Shippey, David Bowes, Bruce Christianson, and Tracy Hall. A mapping study of software code cloning. In *Evaluation Assessment in Software Engineering (EASE 2012), 16th Annual Conference on*, May 2012. (Cited on pages 29, 42 and 49.)

S. Shivaji, E. J. Whitehead, R. Akella, and Kim Sunghun. Reducing features to improve bug prediction. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 600–604. accept, 2009. (Cited on pages 23 and 94.)

Alexander Shvets, Gerhard Frey, and Marina Pavlova. Design patterns explained simply, 2014. URL http://sourcemaking.com/design_patterns/visitor. Accessed 31 August 2014. (Cited on page 58.)

Dag IK Sjoberg, Tore Dyba, and Magne Jorgensen. The future of empirical methods in software engineering research. In *2007 Future of Software Engineering*, pages 358–378. IEEE Computer Society, 2007. (Cited on page 33.)

Dag IK Sjoberg, Aiko Yamashita, Bente Cecilie Dahlum Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156, 2013. (Cited on page 95.)

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083147. URL http://doi.acm.org/10.1145/1082983.1083147. (Cited on pages 1, 7, 13, 21, 37, 39, 43, 45, 46, 47, 48, 53, 55 and 103.)

SpuriousCorrelations. Us spending on science, space, and technology. http://www.tylervigen.com/view_correlation?id=1597, 2014. Accessed 28th July 2014. (Cited on pages ix and 41.)

Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974. (Cited on page 95.)

Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297–310, 2003. (Cited on page 14.)

TheEclipseFoundation. The AspectJ project. http://www.eclipse.org/aspectj, 2014a. Accessed 28th July 2014. (Cited on page 43.)

TheEclipseFoundation. Eclipse - The Eclipse foundation open source community website. http://www.eclipse.org, 2014b. Accessed 24th June 2014. (Cited on page 43.)

TheEclipseFoundation. Eclipse Java development tools (JDT) - overview. http://www.eclipse.org/jdt/overview.php, 2014c. Accessed 24th June 2014. (Cited on page 43.)

Burak Turhan, Tim Menzies, Ayşe Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009. (Cited on pages 1, 16, 90 and 94.)

U.S.NLM. What is genetic testing? http://ghr.nlm.nih.gov/handbook/testing/genetictesting, May 2014. Accessed 6 June 2014. (Cited on page 25.)

Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 408–418, May 2003. doi: 10.1109/ICSE.2003.1201219. (Cited on pages 37 and 46.)

Vera Wahler, Dietmar Seipel, Jürgen Wolff V. Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *In SCAM*, pages 128–135, 2004. (Cited on pages 35 and 98.)

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009. (Cited on pages 66, 97 and 98.)

Gerald E Welch II and Steven G Gabbe. Review of statistics usage in the American Journal of Obstetrics and Gynecology. *American Journal of Obstetrics and Gynecology*, 175(5):1138–1141, 1996. (Cited on page 40.)

Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Adapting a fault prediction model to allow widespread usage. In *Proceedings of the Second International Promise Workshop*, 2006. (Cited on pages 1 and 20.)

Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010. (Cited on page 1.)

Chadd C. Williams and Jaime Spacco. SZZ revisited: verifying when changes induce fixes. In *DEFECTS*, pages 32–36, 2008. (Cited on pages 37 and 46.)

Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011. (Cited on pages 37, 48 and 49.)

Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, 2013. (Cited on page 95.)

John M Yancey. Ten rules for reading clinical research reports. *The American Journal of Surgery*, 159 (6):533–539, 1990. (Cited on page 40.)

Wuu Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21:739–755, June 1991. ISSN 0038-0644. doi: 10.1002/spe.4380210706. URL http://dl.acm.org/citation.cfm?id=116633.116643. (Cited on page 98.)

Andreas Zeller, Thomas Zimmermann, and Christian Bird. Failure is a four-letter word: a parody in empirical research. In *PROMISE*, 2011. ISBN 978-1-4503-0709-3. doi: http://doi.acm.org/10.1145/2020390.2020395. URL http://doi.acm.org/10.1145/2020390.2020395. (Cited on pages 2 and 23.)

Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283, Sept 2009. doi: 10.1109/ICSM.2009.5306304. (Cited on pages 14 and 29.)

Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23, 2011. ISSN 1532-0618. doi: 10.1002/smr.521. URL http://dx.doi.org/10.1002/smr.521. (Cited on page 29.)

Yuming Zhou, Baowen Xu, and Hareton Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *J. Syst. Softw.*, 83(4):660–674, April 2010. ISSN 0164-1212. doi: 10.1016/j.jss.2009.11.704. URL http://dx.doi.org/10.1016/j.jss.2009.11.704. LOC. (Cited on page 1.)

Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, May 2007. (Cited on pages xiii, 37, 43, 46, 49, 50, 51, 52 and 55.)

Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM. (Cited on pages 49, 85, 90 and 91.)

Appendix A

Appendix

A.1 Java Kinds

Java has 92 different kinds located in Tree.Kind. These kinds make up the AST when parsed.

Table A.1: The 92 Java Kinds located in the Tree.Kind and their descriptions

Kind	Description
AND	Used for instances of BinaryTree representing bitwise and logical "and" &.
AND_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing bitwise and logical "and" assignment &=.
ANNOTATION	Used for instances of AnnotationTree.
ARRAY_ACCESS	Used for instances of ArrayAccessTree.
ARRAY_TYPE	Used for instances of ArrayTypeTree.
ASSERT	Used for instances of AssertTree.
ASSIGNMENT	Used for instances of AssignmentTree.
BITWISE_COMPLEMENT	Used for instances of UnaryTree representing bitwise complement operator .
BLOCK	Used for instances of BlockTree.
BOOLEAN_LITERAL	Used for instances of LiteralTree representing a boolean literal expression of type boolean.
BREAK	Used for instances of BreakTree.
CASE	Used for instances of CaseTree.
CATCH	Used for instances of CatchTree.
CHAR_LITERAL	Used for instances of LiteralTree representing a character literal expression of type char.
CLASS	Used for instances of ClassTree.
COMPILATION_UNIT	Used for instances of CompilationUnitTree.
CONDITIONAL_EXPRESSION	Used for instances of ConditionalExpressionTree.
CONDITIONAL_AND	Used for instances of BinaryTree representing conditional-and &&.
CONTINUE	Used for instances of ContinueTree.
CONDITIONAL_OR	Used for instances of BinaryTree representing conditional-or ||.
DIVIDE_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing division assignment /=.
DIVIDE	Used for instances of BinaryTree representing division /.
DOUBLE_LITERAL	Used for instances of LiteralTree representing a floating-point literal expression of type double.
DO_WHILE_LOOP	Used for instances of DoWhileLoopTree.
ENHANCED_FOR_LOOP	Used for instances of EnhancedForLoopTree.

Table A.1 – continued on next page

Table A.1 – continued from previous page

Kind	Description
EMPTY_STATEMENT	Used for instances of EmptyStatementTree.
ERRONEOUS	Used for instances of ErroneousTree.
EQUAL_TO	Used for instances of BinaryTree representing equal-to ==.
EXTENDS_WILDCARD	Used for instances of WildcardTree representing an extends bounded wildcard type argument.
EXPRESSION_STATEMENT	Used for instances of ExpressionStatementTree.
FOR_LOOP	Used for instances of ForLoopTree.
FLOAT_LITERAL	Used for instances of LiteralTree representing a floating-point literal expression of type float.
IDENTIFIER	Used for instances of IdentifierTree.
IF	Used for instances of IfTree.
GREATER_THAN	Used for instances of BinaryTree representing greater-than >.
GREATER_THAN_EQUAL	Used for instances of BinaryTree representing greater-than-equal >=.
INT_LITERAL	Used for instances of LiteralTree representing an integral literal expression of type int.
LABELED_STATEMENT	Used for instances of LabeledStatementTree.
IMPORT	Used for instances of ImportTree.
INSTANCE_OF	Used for instances of InstanceOfTree.
LESS_THAN	Used for instances of BinaryTree representing less-than <.
LESS_THAN_EQUAL	Used for instances of BinaryTree representing less-than-equal <=.
LEFT_SHIFT	Used for instances of BinaryTree representing left shift <<.
LEFT_SHIFT_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing left shift assignment <<=.
MEMBER_SELECT	Used for instances of MemberSelectTree.
METHOD	Used for instances of MethodTree.
LOGICAL_COMPLEMENT	Used for instances of UnaryTree representing logical complement operator !.
LONG_LITERAL	Used for instances of LiteralTree representing an integral literal expression of type long.
MODIFIERS	Used for instances of ModifiersTree.
MINUS_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing subtraction assignment -=.
MINUS	Used for instances of BinaryTree representing subtraction -.
METHOD_INVOCATION	Used for instances of MethodInvocationTree.
NEW_CLASS	Used for instances of NewClassTree.
NEW_ARRAY	Used for instances of NewArrayTree.
MULTIPLY_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing multiplication assignment *=.
MULTIPLY	Used for instances of BinaryTree representing multiplication *.
OR_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing bitwise and logical "or" assignment |=.
OR	Used for instances of BinaryTree representing bitwise and logical "or" |.
NULL_LITERAL	Used for instances of LiteralTree representing the use of null.
NOT_EQUAL_TO	Used for instances of BinaryTree representing not-equal-to !=.
PLUS	Used for instances of BinaryTree representing addition or string concatenation +.
PARENTHESIZED	Used for instances of ParenthesizedTree.

Table A.1 – continued on next page

Table A.1 – continued from previous page

Kind	Description
PARAMETERIZED_TYPE	Used for instances of ParameterizedTypeTree.
OTHER	An implementation-reserved node.
PREFIX_INCREMENT	Used for instances of UnaryTree representing prefix increment operator ++.
PRIMITIVE_TYPE	Used for instances of PrimitiveTypeTree.
REMAINDER	Used for instances of BinaryTree representing remainder %.
REMAINDER_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing remainder assignment %=.
PLUS_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing addition or string concatenation assignment +=.
POSTFIX_DECREMENT	Used for instances of UnaryTree representing postfix decrement operator –.
POSTFIX_INCREMENT	Used for instances of UnaryTree representing postfix increment operator ++.
PREFIX_DECREMENT	Used for instances of UnaryTree representing prefix decrement operator –.
SUPER_WILDCARD	Used for instances of WildcardTree representing a super bounded wildcard type argument.
SWITCH	Used for instances of SwitchTree.
SYNCHRONIZED	Used for instances of SynchronizedTree.
THROW	Used for instances of ThrowTree.
RETURN	Used for instances of ReturnTree.
RIGHT_SHIFT	Used for instances of BinaryTree representing right shift >>.
RIGHT_SHIFT_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing right shift assignment.
STRING_LITERAL	Used for instances of LiteralTree representing a string literal expression of type String.
UNBOUNDED_WILDCARD	Used for instances of WildcardTree representing an unbounded wildcard type argument.
UNARY_PLUS	Used for instances of UnaryTree representing unary plus operator +.
UNSIGNED_RIGHT_SHIFT_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing unsigned right shift assignment
UNSIGNED_RIGHT_SHIFT	Used for instances of BinaryTree representing unsigned right shift >>>.
TYPE_CAST	Used for instances of TypeCastTree.
TRY	Used for instances of TryTree.
UNARY_MINUS	Used for instances of UnaryTree representing unary minus operator -.
TYPE_PARAMETER	Used for instances of TypeParameterTree.
WHILE_LOOP	Used for instances of WhileLoopTree.
VARIABLE	Used for instances of VariableTree.
XOR_ASSIGNMENT	Used for instances of CompoundAssignmentTree representing bitwise and logical "xor" assignment ≙.
XOR	Used for instances of BinaryTree representing bitwise and logical "xor" ˆ.

A.2 Java Code Snippets Significantly Associated with Faults Across All Five Systems

The table below shows the 201 code snippets that are significantly associated with faults across all five systems analysed.

Table A.2: The 201 code snippets that are significantly associated with faults across all five systems analysed

Code Snippet
NULL_LITERAL; BLOCK
NULL_LITERAL; BLOCK; EXPRESSION_STATEMENT
BLOCK; VARIABLE; MODIFIERS; IDENTIFIER
BLOCK; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
BLOCK; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION
LOGICAL_COMPLEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; BLOCK
NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL
BLOCK; EXPRESSION_STATEMENT
CONDITIONAL_AND; METHOD_INVOCATION; MEMBER_SELECT
BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION
BLOCK; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT
BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT
NOT_EQUAL_TO; IDENTIFIER
BLOCK; VARIABLE
NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL; BLOCK
BLOCK; VARIABLE; MODIFIERS
CONDITIONAL_AND; METHOD_INVOCATION
VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION
VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION
VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
RETURN; EXPRESSION_STATEMENT
VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT
PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION
MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER
METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION
PARENTHESIZED; METHOD_INVOCATION
METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL; BLOCK
METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER
MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER

Table A.2 – continued on next page

Table A.2 – continued from previous page

Code snippet
METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; BLOCK
PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT
METHOD_INVOCATION; IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT
PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
METHOD_INVOCATION; IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
PARENTHESIZED; NOT_EQUAL_TO
MODIFIERS; IDENTIFIER; METHOD_INVOCATION
PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL
MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT
MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION
PARENTHESIZED; CONDITIONAL_AND
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; METHOD_INVOCATION
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IDENTIFIER; IF
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER
IDENTIFIER; NULL_LITERAL
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; BLOCK
IDENTIFIER; NULL_LITERAL; BLOCK
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER
IDENTIFIER; NULL_LITERAL; BLOCK; EXPRESSION_STATEMENT
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER
IF; PARENTHESIZED
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IF
IF; PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IF
IF; PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; VARIABLE
IF; PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE
IF; PARENTHESIZED; NOT_EQUAL_TO
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT
IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER
IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED
IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL; BLOCK
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED
IF; PARENTHESIZED; CONDITIONAL_AND
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT

Table A.2 – continued on next page

Table A.2 – continued from previous page

Code snippet
IF; PARENTHESIZED; CONDITIONAL_AND; METHOD_INVOCATION; MEMBER_SELECT
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IF; PARENTHESIZED
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; VARIABLE; MODIFIERS
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; BLOCK
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER
IF; PARENTHESIZED; CONDITIONAL_OR
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; PARENTHESIZED
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER
IDENTIFIER; IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT
IDENTIFIER; IDENTIFIER; BLOCK
IDENTIFIER; BLOCK; EXPRESSION_STATEMENT
IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT
IDENTIFIER; METHOD_INVOCATION
IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION
IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; PARENTHESIZED; TYPE_CAST; IDENTIFIER
IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT; ASSIGNMENT
IDENTIFIER; BLOCK; VARIABLE
IDENTIFIER; IDENTIFIER; BLOCK; IF
IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; IDENTIFIER; BLOCK; IF; PARENTHESIZED
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; IDENTIFIER; BLOCK; VARIABLE
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT
IDENTIFIER; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER

Table A.2 – continued on next page

Table A.2 – continued from previous page

Code snippet
IDENTIFIER; IF
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; IF; PARENTHESIZED
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; PARENTHESIZED; TYPE_CAST
IDENTIFIER; IF; PARENTHESIZED; NOT_EQUAL_TO
IDENTIFIER; METHOD_INVOCATION; IDENTIFIER
IDENTIFIER; IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER
IDENTIFIER; BLOCK; EXPRESSION_STATEMENT; METHOD_INVOCATION
IDENTIFIER; IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL
IDENTIFIER; BLOCK; VARIABLE; MODIFIERS
IDENTIFIER; IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL; BLOCK
IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION
IDENTIFIER; IF; PARENTHESIZED; CONDITIONAL_AND
IDENTIFIER; IF; PARENTHESIZED; CONDITIONAL_AND; NOT_EQUAL_TO
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; BLOCK
IDENTIFIER; IDENTIFIER; IF
IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT
IDENTIFIER; IDENTIFIER; METHOD_INVOCATION
MEMBER_SELECT; IDENTIFIER; VARIABLE
IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER
MEMBER_SELECT; IDENTIFIER; VARIABLE; MODIFIERS
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED; NOT_EQUAL_TO
MEMBER_SELECT; IDENTIFIER; BLOCK
IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT
MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE
IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT
MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS
IDENTIFIER; IDENTIFIER; IDENTIFIER; IF
MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED; METHOD_INVOCATION
MEMBER_SELECT; IDENTIFIER; BLOCK; VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER; NULL_LITERAL
MEMBER_SELECT; METHOD_INVOCATION
IDENTIFIER; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER
MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER
IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER; BLOCK
IDENTIFIER; IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT

Table A.2 – continued on next page

Table A.2 – continued from previous page

Code snippet
IDENTIFIER; EXPRESSION_STATEMENT
IDENTIFIER; IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED
IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED
IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED; NOT_EQUAL_TO; IDENTIFIER
IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED; CONDITIONAL_OR
IDENTIFIER; EXPRESSION_STATEMENT; ASSIGNMENT
IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT
IDENTIFIER; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
MEMBER_SELECT; IDENTIFIER; IF
MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER
MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER
EXPRESSION_STATEMENT
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IF
IF
MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
PARENTHESIZED
MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION
EQUAL_TO
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IDENTIFIER; IF
NOT_EQUAL_TO
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; BLOCK
CONDITIONAL_AND
MEMBER_SELECT; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT
NULL_LITERAL
MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT
EXPRESSION_STATEMENT; METHOD_INVOCATION
MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT
MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
MEMBER_SELECT; IDENTIFIER; IDENTIFIER
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; IF; PARENTHESIZED
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT
MEMBER_SELECT; IDENTIFIER; IDENTIFIER; BLOCK; EXPRESSION_STATEMENT
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER

Table A.2 – continued on next page

Table A.2 – continued from previous page

Code snippet
MEMBER_SELECT; IDENTIFIER; IF; PARENTHESIZED
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER; IDENTIFIER
MEMBER_SELECT; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER
MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT
MEMBER_SELECT; IDENTIFIER; EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT