# Some Code Smells Have a Significant but Small Effect on Faults

TRACY HALL, Brunel University
MIN ZHANG, DAVID BOWES, and YI SUN, University of Hertfordshire

We investigate the relationship between faults and five of Fowler et al.'s least-studied smells in code: Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man. We developed a tool to detect these five smells in three open-source systems: Eclipse, ArgoUML, and Apache Commons. We collected fault data from the change and fault repositories of each system. We built Negative Binomial regression models to analyse the relationships between smells and faults and report the McFadden effect size of those relationships. Our results suggest that Switch Statements had no effect on faults in any of the three systems; Message Chains increased faults in two systems; Message Chains which occurred in larger files reduced faults; Data Clumps reduced faults in Apache and Eclipse but increased faults in ArgoUML; Middle Man reduced faults only in ArgoUML, and Speculative Generality reduced faults only in Eclipse. File size alone affects faults in some systems but not in all systems. Where smells did significantly affect faults, the size of that effect was small (always under 10 percent). Our findings suggest that some smells do indicate fault-prone code in some circumstances but that the effect that these smells have on faults is small. Our findings also show that smells have different effects on different systems. We conclude that arbitrary refactoring is unlikely to significantly reduce fault-proneness and in some cases may increase fault-proneness.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*

General Terms: Experimentation, Languages, Design

Additional Key Words and Phrases: Software code smells, defects

**ACM Reference Format:**
Tracy Hall, Min Zhang, David Bowes, and Yi Sun. 2014. Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 33 (August 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2629648

## 1. INTRODUCTION

The term 'Bad Smells in Code' was used by Beck [Fowler and Beck 1999] to describe 22 particular structures in code that can cause detrimental effects on software and should be refactored. Yet, Fowler and Beck [1999] are not specific about the detrimental effects that smells cause. Many smells have not been studied, and we have no evidence about the nature or size of the detrimental effects of these smells. Consequently, it is hard for developers to determine the importance of each smell.

The aim of this article is to identify the effects of smells on faults. Evidence on these effects is important to practitioners and to researchers. Practitioners' fault reduction effort can be directed to refactoring those smells most likely to be indicators of fault-prone code. Practitioners' fault reduction effort need not be wasted on code containing smells that have no effect on faults. Researchers can focus their effort on developing



Authors' addresses: T. Hall (corresponding author), Brunel University; email: tracy.hall@brunel.ac.uk; M. Zhang, D. Bowes, and Y. Sun, University of Hertfordshire.

DOI: http://dx.doi.org/10.1145/2629648

effective detection and elimination strategies and tools for those smells that most affect faults.

The effects of some smells have been studied previously. For example, Li and Shatnawi [2007] studied the relationship between six smells and faults.[1] Olbrich et al. [2009, 2010] studied three smells[2] in relation to the evolution of systems. The size of these effects have not generally been measured except by Sjøberg et al. [2013], who studied the effect of 12 smells on maintenance effort. We investigate five of Fowler et al.'s least-studied smells (Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man). We are the first to investigate the effect of these five smells on software faults and to measure the size of these effects.

No existing tool detects all of the five smells that we investigate, and so we developed our own detection tool. This tool identifies code representing each of the five smells and determines whether or not any of these smells occur in a Java file. We used version control and fault repository data to identify the occurrence of faults in these Java files. We collected smell and fault data from Eclipse, ArgoUML, and Apache Commons open-source projects. We examined the distribution of the smell and fault data in these three projects. This examination enabled us to identify negative binomial regression as the most appropriate modelling technique for this data. We developed a negative binomial regression model, where the dependent variable is a count of the number of faults in a file. The independent variables in the model are the occurrence of Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man smells in a file. An additional independent variable is the number of lines of code (LOC) in a file. This additional variable is to investigate file size as a possible confounding factor. We calculated the McFadden effect size of the significant relationships identified by our model.

We make the following five contributions in this article. First, we identify Switch Statements as unlikely to affect faults. Second, we show that smells can effect faults in some systems but not all systems. Third, Message Chains increase faults in some systems. Fourth, code size impacts the effect that some smells have on faults. Fifth, where smells do affect faults the effect size is small.

The remainder of this article is structured as follows: Section 2 presents a brief summary of the background to smells. Section 3 describes the methods applied in this paper. Section 4 presents the results of our study. Section 5 discusses the results we report. Section 6 identifies threats to the validity of the study. Finally, conclusions are drawn in Section 7.

## 2. CODE SMELLS

### 2.1. Introduction to the Smells

Many code smells have been identified over the years. Brown et al. [1998] introduced the idea of code structures in the form of 40 anti-patterns. Probably the best known of these anti-patterns is Spaghetti Code, where the code structure is incomprehensible. Kerievsky [2004] also identified many other structures that should be refactored from code. Beck and Fowler identified the most widely known set of 22 smells [1999]. These smells cover a wide range of software problems. For example, the Switch Statements smell is a simple code structure that may indicate code duplication. The Shotgun Surgery smell is a complicated code structure, a change to which affects a sequence of different modules. Fowler and Beck [1999] say that these 22 smells 'give you indications that there is trouble that can be solved by a refactoring'. Mens and Tourwe [2004]

[1]Large Class, Long Method, Shotgun Surgery, Data Class, Refused Bequest, and Feature Envy.
[2]God Class, Shotgun Surgery and Brain Class.

suggest that code refactoring is most commonly directed by Fowler et al.'s smells. Some studies report relationships exist between smells. For example, Yamashita and Moonen [2013a] found relationships between God Method and God Class smells, and a relationship between Data Clumps and Data Class—not unexpected relationships given the similarity of these smells.

Our previous systematic literature review [Zhang et al. 2011] of all 39 studies of smells published between 2000–2009 revealed that some of Fowler et al.'s 22 smells have been studied more than others. Yamashita [2012] also report similar patchy coverage of smells in published studies. For example, our recent mapping study [Shippey et al. 2012] shows that Clones (or Duplicated Code) have been studied many times (e.g., [Gode and Koschke 2011; Rahman et al. 2010]). Shotgun Surgery, God Classes, and God Methods have been studied several times (e.g., [Sjøberg et al. 2013; Olbrich et al. 2009, 2010; Vaucher et al. 2009; Shatnawi and Li 2006; Li and Shatnawi 2007]. Many smells have been studied very little, for example, Feature Envy [Oliveto et al. 2011; Li and Shatnawi 2007]. Our review [Zhang et al. 2011] shows that 12 of Fowler's 22 smells have been investigated only within general studies of smells. These 12 smells have not been studied in detail or in isolation, having been studied alongside all other smells within the same five general studies [Counsell et al. 2006; Mäntylä et al. 2003, 2004; Mäntylä and Lassenius 2006; Trifu and Reupke 2007].

The reasons for the lack of research attention are difficult to establish but seem to be partially related to the ease with which smells can be automatically identified in code. For example, Li and Shatnawi [2007] say that ease motivated their choice of smells. Those smells that have received most attention (e.g., Duplicated Code, Long Method, and Large Class [Zhang et al. 2011]) are relatively easy to automatically identify in code. Those receiving least attention seem relatively more difficult to automatically identify. Ease of identification does not fully explain variation in research attention. For example, neither the Switch Statement nor the Comments smell has been the focus of many studies, although both smells seem relatively easy to automatically identify in code. Alongside ease of identification, researchers may also be selecting smells that seem most important to study, are most widely known (e.g., Olbrich et al. [2009]) or which occur most commonly in code.

Establishing evidence that identifies the impact (or not) of all smells is important, because such evidence will help developers to determine the importance of individual smells. Developers can then make more informed decisions on the prioritisation of refactorings. Consequently, we focus this study on those smells of which least is currently known about their effect.

### 2.2. Effects of Smells on Code

Several previous studies investigate the impact that smells have on the evolution of code. Khomh et al. [2009a] report that classes containing smells are changed more frequently than other classes. Olbrich et al. [2009] showed that components 'infected' by smells need more maintenance and exhibit different change behaviour. Du Bois et al. [2006] found that refactoring God Classes improves code comprehensibility. The negative effect smells have on faults has also been studied previously. Li and Shatnawi analysed six smells in relation to faults [2007; Shatnawi and Li 2006]. They reported that Shotgun Surgery, God Class, and God Method smells are associated with higher levels of fault-proneness.

Studies also show that some smells are not necessarily bad and may be good. Li and Shatnawi [2007] found that Refused Bequest, Feature Envy, and Data Class had no impact on fault-proneness. Sjøberg et al. [2013] found that Refused Bequest was significantly associated with decreased maintenance effort. Monden et al. [2002]

report that a limited amount of Duplicated Code increases the reliability of software. Kapser and Godfrey [2008] indicate that Duplicated Code may not be harmful in many situations. Rahman et al. [2010] found a relationship between Duplicated Code and reduced fault-proneness. D'Ambros et al. [2010] found no relationship between faults and Feature Envy or Shotgun Surgery. Abbes et al. [2011] report that a single anti-pattern in code does not reduce code comprehension (though more than one anti-pattern in the same code does impede comprehension). Khomh et al. [2009b] report that developers do not always believe that the use of patterns improves the quality of code.

Other studies identify the importance of size in relation to smells. Sjøberg et al. [2013] found that once their results were adjusted for file size and number of changes, none of the 12 smells that they investigated were associated with increased maintenance effort. Yamashita and Counsell [2013] also report that code smells are influenced by code size. Olbrich et al. [2010] show the impact that God and Brain Class smells have on code is directly influenced by class size. Classes in which these smells occurred were initially related to higher numbers of faults. However, once those classes were normalised for size, they became related to lower numbers of faults. Vokac [2004] similarly reports results where faults in classes participating in the Singleton and Observer patterns are very sensitive to size. Zhou et al. [2009] also demonstrated that class size confounded the relationship between some object-oriented metrics and change to code. These findings suggest that size is an important confounding factor. We have taken size into account in this study.

### 2.3. Detection of Smells

Various approaches to detecting smells have been developed. Early detection approaches tend to be manual, for example, Travassos et al. [1999] describe an approach based on reading code. Mäntylä et al. [2004] showed that the manual detection of smells has limitations in terms of resources, scalability, and objectivity. In a follow-up study, Schumacher et al. [2010] reported that humans are not good at detecting God Classes—automated approaches are more competitive. Automated smell detection approaches include the use of metrics. Munro [2005] proposed a metrics-based approach to smell detection based on formalising each smell then identifying heuristics that can be implemented using metrics. Marinescu [2004] also developed a metrics-based approach to smell detection, as have Rao and Reddy [2008]. Pattern detection has also underpinned many smell detection studies. For example, the DeMIMA approach [Guéhéneuc and Antoniol 2008] to identifying design patterns has been used in studies (e.g., [De Lucia et al. 2010]). Machine learning approaches are also increasingly used to detect smelly code and to generate detection rules. For example, Fontana et al. [2013] report promising results using six machine learning techniques, Boussaa et al. [2013] report on their early work using competitive coevolutionary search, and Khomh et al. [2009b] use a Bayesian-based approach to detect God Classes. Other detection approaches have also been developed. For example, Liu et al. [2013] propose resolution sequences to detect smells, and Palomba et al. [2013] report very good detection results based on using a combination of change history data and code structure information.

Moha et al. [2010] provide the comprehensive DECOR approach to specifying and detecting smells. DECOR detects well-known, predefined smells, and also provides a domain-specific language allowing for the definition of other smells, based on code metrics as well as structural and lexical code properties. A detection technique that instantiates this method, called DETEX, is also described by Moha et al. [2010]. This extensibility potentially provides flexible and useful code smell detection, though is yet to be used in many published studies.

### 2.4. Detection Tools

Automated tools make it possible to consistently detect smells. Tools also allow smells to be detected and analysed in very large code bases. Consequently a range of closed and open-source smell detection tools have been developed. Two closed-source tools resulted from the work of Marinescu (Borland Together[3] and InCode[4]). These tools have been used in several published studies (e.g., Sjøberg et al. [2013]). Closed-source tools have the limitation that it is not usually possible to examine the tool's code to establish the smell detection strategy implemented.

Open-source tools have the advantage that it is possible to examine the detection strategies implemented by the tool. Many such tools have been developed, including JDeodrant[5], PMD[6], iPlasma[7] [Marinescu et al. 2005], InFusion[8], and StenchBlossom[9] [Murphy-Hill and Black 2010a]. In addition, Ptidej[10] [Guéhéneuc 2005] is a tool developed to automate the DECOR smell detection approach. Most recently the monitoring and instant refactoring tool InsRefactor has been presented by Liu et al. [2013].

Although there are a wide variety of tools available, each tool detects only a subset of smells. No tool is preset to detect all smells. There is relatively small overlap in the smells the tools detect. No tool detects all the smells that we investigated. Few tools detect any of the five smells that we investigate. Although Ptidej (via DECOR) provides potential extensibility to all of the smells we investigate, our evaluation [Bowes et al. 2013] showed that it was not possible to extend DECOR to all five smells that we were investigating. This made the Ptidej tool (and DECOR) not suitable for our study (discussed in more detail in Section 3.4). Consequently, we developed our own detection tool (described in the next section).

## 3. METHODS

### 3.1. Aims of the Investigation

The aim of this study is to investigate whether, taking into account file size, the five smells targeted in this investigation are more likely to be related to fault-prone files than files not containing these smells. Consequently, we test the following hypotheses at the $p<0.05$ level of significance.

*Hypotheses* 1. The Data Clumps Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.

*Hypotheses* 2. The Middle Man Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.

*Hypotheses* 3. The Speculative Generality Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.

*Hypotheses* 4. The Switch Statements Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.

[3]http://www.borland.com/products/together.
[4]http://www.intooitus.com/products/incode.
[5]http://www.jdeodorant.com/.
[6]http://pmd.sourceforge.net/.
[7]http://loose.upt.ro/reengineering/research/iplasma.
[8]http://www.intooitus.com/products/infusion.
[9]http://people.engr.ncsu.edu/ermurph3/tools.html.
[10]http://www.ptidej.net/tool/.

Table I. Summary of the Five Smells Investigated

| Smell name | Fowler et al. definitions |
|---|---|
| Data Clumps | Some data items together in lots of places: fields in a couple of classes, parameters in many method signatures. |
| Message Chains | You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on. Navigating this way means the client is coupled to the structure of the navigation. Any change to the intermediate relationships causes the client to have to change. |
| Middle Man | You look at a class's interface and find half the methods are delegating to this other class. It may mean problems. |
| Speculative Generality | If the machinery were being used, it would be worth it. But if it isn't, it isn't. The machinery just gets in the way, so get rid of it. |
| Switch Statements | Switch statements often lead to duplication. Most times you see a switch statement you should consider polymorphism. |

*Hypotheses* 5. The Message Chain Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.

*Hypothesis* 6. The size of files is not related to numbers of faults in files.

### 3.2. Smells Investigated

The five smells that we investigated are described in Table I. We selected these five smells from the 12 which have not had their effect previously investigated (mentioned in Section 2). We chose these five smells from the 12 on the basis that they were the most straightforward to automatically detect. To automatically detect smells, they must be well defined. Fowler et al.'s definitions vary in their preciseness. From the 12 least-studied smells, we identified those that are most precisely defined. We rated the preciseness of the definition given on a three point scale (described in detail in Zhang et al. [2008]). We selected the six most precisely defined smells as those most suitable for automatic detection. From this six, the Parallel Inheritance Hierarchies smell was then eliminated, as, unlike the other five, its identification is dependent on multiple versions of code.

### 3.3. Measuring Smells

We detect two of the five smells investigated (Switch Statements and Message Chains) as a count value for each file (i.e., we measure these smells continuously). This integer represents the number of times the smell has occurred in that file. Such integer measurement is usually applied to smells based at the method level. We detect the remaining three smells (Data Clumps, Speculative Generality, and Middle Man) as a binary value for each file (i.e., we measure these smells categorically). This binary value represents whether a particular smell is present or not in the file. Such binary measurement is usually applied to smells based at the class level. Our combination of binary and integer smell measurement is not unusual, for example, Li and Shatnawi [2007] also measured two of the six smells that they investigated as integers.

### 3.4. Smell Detection

We developed our own tool to automatically detect the five targeted smells in Java source code for several reasons.

(1) None of the existing smell detection tools, of which we are aware, detect the five smells that we are investigating.

(2) No ideal tool already exists. There is evidence to suggest that existing tools are not consistent in the smells they detect. The Fontana et al. [2011] evaluation of five existing smell detection tools shows that tools inconsistently detect smelly code (i.e., each tool detects different code as containing the same smell). Our more detailed study further confirms Fontana et al.'s findings [Bowes et al. 2013].
(3) There are currently no commonly accepted and used definitions of smells. Smells are subjective. Universally defining and detecting smells is increasingly being recognised as 'challenging, if not impossible' [Liu et al. 2013]. Therefore not surprisingly, the definitions used by tools for the same smell vary. Smell definitions seem to be based on researchers' informal and personal interpretations of Fowler et al.'s initial descriptions. Often, it is difficult to identify the smell definitions being used by tools as these definitions are not usually documented and must be reverse-engineered from the code (assuming the code is open source). In particular the thresholds which tools use to identify a code structure as a smell or not vary from one tool to another. For example, the thresholds used to define how long a Message Chain must be for it to be classified as a Message Chain varies from one tool to another [Bowes et al. 2013]. Variation in the thresholds used by tools partially explain why tools inconsistently detect smelly code. Using our own tool has the advantage that we know the definitions and thresholds that we are using and these definitions are documented and publicly available for future users of our tool.
(4) As our previous evaluations demonstrate [Randall 2012; Bowes et al. 2013], the smell definition and detection extensibility of DECOR proved impossible to implement for all five of our smells. Our evaluation, found DECOR to be technically challenging to extend and did not, at the time, support the detection of four of the five smells we investigated. The main reason that DECOR did not support the smells we were investigating is that Ptidej did not collect intra-method relationship data; instead Ptidej collects inter-method relationship data (not helpful for detecting our smells). In addition, DECOR did not support the detection of Duplicated Code (Ptidej currently does not appear to have the facility to select or analyse code sections), Switch Statements (for the same reasons as Duplicated Code), Middle Man, Data Clumps, or Speculative Generality (it is possible only to implement a very narrow definition currently in Ptidej). For a full discussion of the smells DECOR does and does not support, see [Randall 2012]. The current limitations of DECOR for the smells we investigated are confirmed by our private correspondence with the DECOR team. Message Chains was the only smell of our five that DECOR did detect. Our evaluation found that DECOR performed particularly well in detecting Message Chains, which were technically difficult to detect and not easily observable by humans [Bowes et al. 2013].

Our tool is based on detecting code representative of each smell. This approach is based on that used by Mens et al. [2003]. We used the open-source API Recoder[11] to transform Java source code into abstract syntax trees. We then developed detection algorithms to search the syntax trees using our definitions of the five targeted smells. Our approach includes the following three steps.

*Step One: Defining the Smells.* It is essential that each smell is defined sufficiently precisely to consistently and reliably detect that smell in a piece of code. From Fowler et al.'s definitions, we specified code templates representative of the five targeted smells. We validated and refined our code templates using a panel of four practitioner

[11]Recoder API can be found at `http://sourceforge.net/apps/mediawiki/recoder/index.php?title=Main_Page` (last checked 2013-07-30).

Table II. Definition of Data Clump

| | |
|---|---|
| **Fowler et al.'s definition** | Data items hang around in groups. Often you will see the same three or four data items together in lots of places: fields in a couple of classes, parameters in many method signatures. |
| **Our definition** | **Occurrence 1:**<br>1. Three or more data fields stay together in more than one class.<br>2. These data fields should have same signatures (same names, same data types, and same access modifiers).<br>3. These data fields may not group together in the same order.<br>**Occurrence 2:**<br>1. Three or more input parameters stay together in more than one method's declaration.<br>2. These parameters should have same signature (same names, same data types).<br>3. These parameters may not be grouped together in the same order.<br>4. These methods should not be in the same inheritance hierarchy and with the same method signature.<br>NB: A Data Clump can be two integers and one float in Place A, and the same two integers and one float in Place B. It should not be two integers and one float in Place A, and one integer, one boolean and one float in Place B. |

Table III. Definition of the Message Chains

| | |
|---|---|
| **Fowler et al.'s definition** | You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on. You may see these as a long line of getThis methods, or as a sequence of temps. |
| **Our definition** | **Occurrence 1:**<br>1. In order to access a data field in another class, a statement must call more than a threshold value of getter methods in a sequence (e.g., int a=b.getC().getD(); or int a=b.getD(c.getC());).<br>2. This method call statement and the declarations of getter methods are in different classes.<br>**Occurrence 2:**<br>1. A method has more than a threshold number of temporary variables.<br>2. A temporary variable (an object) is a variable that only accesses data members (data fields/getter methods) of the other classes or other temp variables.<br>NB: Message Chains for any class which is imported, and for which we have the source code, have been followed. Message Chains are not followed for any imports from a jar file for which we do not have the source code. This means that Message Chains that result from import statements (e.g., using Java Libraries) are excluded as their use is beyond the control of developers. |

Table IV. Definition of the Middle Man

| | |
|---|---|
| **Fowler et al.'s definition** | You look at a class's interface and find half the methods are delegating to this other class. |
| **Our definition** | 1. At least half of a class's methods are delegation methods.<br>2. A delegation method is a method that<br>(a) Contains at least one reference to another Class.<br>(b) Contains less than a threshold value of LOC. |

and academic experts (the process of validating our definitions are reported in Zhang et al. [2008]). The templates of code that we detect, relative to Fowler et al.'s definitions, are described in Tables II to VI. The thresholds that we applied to these definitions are provided in Table VII.

*Step Two: Transforming Source Code into Abstract Syntax Trees.* We used the open-source API Recoder version 0.84 to translate the source code from Eclipse,

Table V. Definition of the Speculative Generality

| | |
|---|---|
| **Fowler et al.'s definition** | If the machinery were being used, it would be worth it. But if it isn't, it isn't. The machinery just gets in the way, so get rid of it. This kind of machinery includes: abstract classes that aren't doing much, methods with unused parameters, methods named with odd abstract names. |
| **Our definition** | **Occurrence 1:**<br>1. A class is an abstract class or interface.<br>2. This class has not been inherited or is only inherited by one class.<br>**Occurrence 2:**<br>A class contains at least one method that contains at least one parameter which is unused. |

Table VI. Definition of the Switch Statements

| | |
|---|---|
| **Fowler et al.'s definition** | The problem with switch statement is essentially that of duplication. Often you find the same switch statement scattered about a program in different places. If you add a new clause to the switch, you have to find all these switch statements and change them. So most times you see a switch statement you should consider polymorphism. |
| **Our definition** | **Occurrence 1:**<br>1. The code contains an instance of the switch keyword.<br>2. A switch has more than two branches (including default statement).<br>3. Each branch has more than a threshold value of LOC.<br>**Occurrence 2:**<br>1. The code contains an instance of if-else keyword.<br>2. This if-else block has more than two branches.<br>3. Each branch has more than a threshold value of LOC.<br>4. The logic expressions in the if-else statements are type checking expressions using instanceof key words. |

Table VII. Threshold Values for each Code Smell

| **Detection Component** | **Threshold** | | | **Remarks** |
|---|---|---|---|---|
| | **Name** | **Value** | **Rational** | |
| Data Clumps | Minimum instances | 3 | According to Fowler et al.'s definition of Data Clumps Smell. | A Data Clump should contain three or more instances of data fields or input parameters which have same signatures. |
| Message Chains | Minimum length | 2 | According to Fowler et al., the minimum possible length of a Message Chain is two. | A Message Chain should be a chain of method calls with more than two getter method calls or temp variables. |
| Middle Man | Maximum LOC | 3 | A delegation method should only reference to other classes and do not have other functionality. | The delegation methods of a Middle Man should have no more than three lines of code. |
| Speculative Generality | N/A | N/A | N/A | N/A |
| Switch Statements | Minimum branches | 2 | A one branch Switch Statement is not a valid Switch Statement. | A Switch Statement should contain a least two branches. |
| | Minimum LOC | 3 | Fowler et al. suggest that the problem of Switch Statements is they could cause duplication in code. However in our opinion if the size of each branch of a switch statement is too small that duplication is not likely to cause troubles. | Each branch of a Switch Statement has more than three lines of code. |

ArgoUML, and Apache Commons into abstract syntax trees. The Recoder API is a Java framework for source code meta-programming. It provides an infrastructure for Java source code analysis and transformation tools.

*Step Three: Detecting Smells in the Abstract Syntax Trees.* We developed Java detection algorithms for each smell (based on the definitions produced during Step One). These algorithms are based on searching the abstract syntax trees for code containing each of the five smells. The result of this search is that every file is labeled as either containing (or not) each of the three smells that we detect categorically and with a number for the two smells that we detect continuously. Our detection tool is available at `http://sourceforge.net/projects/cbsdetector/`.

### 3.5. Evaluating the Performance of Our Approach to Smell Detection

*3.5.1. An Overview of Our Evaluation Process.* We evaluated the performance of our smell detection tool by comparing the detection performance of our tool with two other detection tools (Stench Blossom [Murphy-Hill and Black 2010b] and DECOR [Moha et al. 2010]). We also manually evaluated the performance of our tool using two humans. We chose to evaluate our tool against Stench Blossom, as, unlike other tools, it detects a relatively high number of the smells that we investigate (three of the five smells). We chose to evaluate our tool against DECOR, as it is a well-known and increasingly-cited extensible smell detection approach. The DECOR method [Moha et al. 2010] reports precision values that vary from one smell to the next (from 41.1% to 88.6%) but consistently report 100% recall values. As far as we know, no definitively superior smell detection tool has emerged against which to compare our tool.

Our tool-based performance evaluation consisted of running our tool, Stench Blossom and DECOR on the whole ArgoUML code base (10,539 methods in 1,582 files). Our tool detected five smells; Stench Blossom detected three smells (Switch Statements, Message Chains and Data Clumps); DECOR detected only one smell (Message Chains). The limitations of the smells DECOR detects are discussed in Section 3.4 and Bowes et al. [2013]. For each smell, we then analysed the detection agreements and disagreements between our tool and the other two tools using Cohen's Kappa statistic [Cohen 1960]. Agreement using this statistic is measured on a scale [−1..1] with the following interpretations used: [−1..0] less than chance agreement; [0.01..0.20] slight agreement; [0.21..0.40] fair agreement; [0.41..0.60] moderate agreement; [0.61..0.80] substantial agreement; [0.81..0.99] almost perfect agreement. We chose to report detection performance using Cohen's Kappa statistic rather than precision or recall, as no reliable smell baseline exists against which tool detections can be compared. It is not possible to definitively measure the detection performance of any tool using a real project like ArgoUML as no exhaustive set of smell data is available. It is not possible to establish absolutely whether a tool has detected smells accurately on such real systems.

Our manual evaluation involved two researchers (DR and TS) inspecting 100 samples of ArgoUML code. For each of the three smells that our tool and Stench Blossom detected, we randomly chose 100 pieces of code that the two tools did not agree contained a particular smell. We based our manual evaluation on these disagreements, as it gave us an opportunity to look in detail at the basis of the disagreements between the tools, added to which it was not practical to manually inspect the whole ArgoUML code base. Depending on the granularity with which we were measuring a particular smell, that is, continuously or categorically, these code samples were either 100 files or 100 methods. The two researchers were experienced Java programmers with a good knowledge of Fowler et al.'s smells. Each was given Fowler et al.'s definitions of the five smells that we were investigating (Shown in Table I). The two researchers then

Table VIII. Switch Statement Detections Made by Our Tool and Stench Blossom in all 10,539 ArgoUML Methods

| | | Stench Blossom | | |
|---|---|---|---|---|
| | | Absent | Present | |
| Our Tool | Absent | 10,331 | 177 | |
| | Present | 17 | 14 | |
| $\kappa = 0.9593$ | | | | $n = 10{,}539$ |

Table IX. Switch Statement Detections Manually Made by Two Human Inspectors in a Sample of ArgoUML Methods (Overall Kappa Scores)

| | | Human Inspectors | | |
|---|---|---|---|---|
| | | DR | TS | |
| Tools | Our Tool | 0.31 | −0.05 | |
| | Stench Blossom | 0.30 | 1.0 | |
| | | | | $n = 100$ |

each independently inspected the code samples and decided whether or not the sample contained the smell that the two tools had disagreed on. For each smell, we again analysed the detection agreements between the tools and the human inspectors using Cohen's Kappa statistic [Cohen 1960]. Further details of our performance evaluations are available from (Randall [2012] and Bowes et al. [2013]).

*3.5.2. The Evaluation Results.* Table VIII shows Switch Statement detections made by both our tool and Stench Blossom in analysing all 10,539 methods in ArgoUML. Table VIII shows that there was almost perfect agreement (0.96 Kappa score) between our tool and Stench Blossom in detecting Switch Statements. DECOR does not detect Switch Statements (as discussed in Section 3.4 and in [Bowes et al. 2013]), so we were unable to compare the performance of our tool in detecting Switch Statements against that of DECOR. We manually investigated a sample of 100 methods from the 4% disagreement between our tool and Stench Blossom in detecting Switch Statements. Table IX shows that the two manual inspectors detected Switch Statements very differently. TS agreed totally with StenchBlossom's detection of Switch Statements and totally disagreed with our tool's detection. We discovered that TS had not applied a threshold as our tool does[12] to identifying a code construct as a Switch Statement. TS identified anything with the keyword 'Switch' as a positive detection. Stench Blossom has a similar threshold-free approach to Switch Statements. Table IX shows that DR has fair agreement with both our tool (0.31 Kappa score) and Stench Blossom (0.30 Kappa score). We discovered that DR was taking a more sophisticated approach to detecting switch statements and also identifying if-then-else constructs as Switch Statements. The two inspectors had only moderate agreement (0.44 Kappa score) between them on Switch Statements.

Table X shows Message Chain detections made by our tool and by Stench Blossom in analysing all 10,539 methods in ArgoUML and shows that there is substantial agreement (0.78 Kappa score) between the two tools in detecting Message Chains. Table XI shows agreement in detecting Message Chains between our tool and DECOR in all 1,582 ArgoUML files.[13] Table XI shows slight agreement (0.15) between these two tools. Table XII shows agreement in detecting Message Chains between Stench

[12]Table VII shows that our threshold for a Switch Statement is that a construct must have at least three branches.

[13]DECOR only detects Message Chains at the file level so we are unable to make detective comparisons with DECOR at the method level.

Table X. Message Chain Detections Made by Our Tool and Stench Blossom in all 10,539 ArgoUML Methods

| | | Stench Blossom | | |
|---|---|---|---|---|
| | | Absent | Present | |
| Our Tool | Absent | 7,766 | 1,830 | |
| | Present | 106 | 837 | |
| $\kappa = 0.7755$ | | | | $n = 10{,}539$ |

Table XI. Message Chain Detections Made by Our Tool and DECOR in 1,582 ArgoUML Files

| | | Our Tool | | |
|---|---|---|---|---|
| | | Absent | Present | |
| DECOR | Absent | 1,039 | 364 | |
| | Present | 87 | 92 | |
| $\kappa = 0.15$ | | | | $n = 1{,}582$ |

Table XII. Message Chain Detections Made by DECOR and Stench Blossom in 1,582 ArgoUML Files

| | | Stench Blossom | | |
|---|---|---|---|---|
| | | Absent | Present | |
| DECOR | Absent | 482 | 921 | |
| | Present | 35 | 144 | |
| $\kappa = 0.05$ | | | | $n = 1{,}582$ |

Blossom and DECOR. Table XII shows lower (slight) agreement (0.05) between these two tools on Message Chains. Overall, the three tools do not agree with each other on what a Message Chain is. We manually investigated this disagreement using a sample of 100 methods from the 21% disagreement between our tool and Stench Blossom in detecting Message Chains. Table XIII shows mixed agreements between the two manual inspectors and the three tools. Fair agreement (0.23) occurred between the two inspectors. Overall tools and people do not substantially agree on what a Message Chain is and they identify different code constructs as Message Chains. One of the differences between the tools is that DECOR identifies transitive calls as a Message Chain. Such transitive calls are not identified by our tool nor by Stench Blossom. In addition, such transitive calls were also not picked up by the human inspectors. This is not surprising, as such calls are hard to identify by visual inspection.

Table XIV shows Data Clump detections made by both our tool and Stench Blossom in analysing all 1,582 ArgoUML files. Table XIV shows slight agreement (0.05 Kappa score) in detecting Data Clumps between our tool and Stench Blossom. DECOR does not detect Data Clumps (as discussed in Section 3.4 and in [Bowes et al. 2013]), so we were unable to compare the performance of our tool in detecting Data Clumps against that of DECOR. Table XV shows that the manual inspectors had extremely poor detection agreements with either tool for Data Clumps. In addition, the two inspectors disagreed with each other on detecting Data Clumps ($\kappa = -0.0556$). Overall agreement levels between tools and between inspectors were lowest for detecting Data Clumps. We believe that this result is due to Data Clumps being particularly difficult to formally define from Fowler et al.'s original definitions. Consequently, tools and people use different definitions in their detections.

We were unable to evaluate the performance of our tool detecting Speculative Generality or Middle Man, as neither of the tools detect these smells. However, given the variable agreements between all three tools on detecting the smells, we have no reason to believe that the results would have been any less variable for the two additional

Table XIII. Message Chain Detections Manually Made by Two Human Inspectors in a Sample ArgoUML Methods (Overall Kappa Scores)

| | | Human Inspectors | | |
|---|---|---|---|---|
| | | DR | TS | |
| Tools | Our Tool | −0.40 | −0.38 | |
| | Stench Blossom | 0.28 | −0.01 | |
| | DECOR | 0.31 | 0.09 | |
| | | | | $n = 100$ |

Table XIV. Data Clump Detections Made by Our Tool and Stench Blossom in all 1,582 ArgoUML Files

| | | Stench Blossom | | |
|---|---|---|---|---|
| | | Absent | Present | |
| Our Tool | Absent | 958 | 523 | |
| | Present | 50 | 51 | |
| $\kappa = 0.0477$ | | | | $n = 1{,}582$ |

Table XV. Data Clump Detections Manually Made by Two Human Inspectors in a Sample of ArgoUML Methods (Overall Kappa Scores)

| | | Human Inspectors | | |
|---|---|---|---|---|
| | | DR | TS | |
| Tools | Our Tool | −0.1 | 0.0 | |
| | Stench Blossom | −0.1 | 0.0 | |
| | | | | $n = 100$ |

smells. Like our results, DECOR [Moha et al. 2010] similarly reports precision values which vary from one smell to the next (from 41.1% to 88.6%).

*3.5.3. Conclusions of Our Tool Performance Evaluation.* Our evaluation shows that it is incredibly hard to define and operationalise smell definitions either for automatic or manual smell detection. Although for some smells our tool agreed well with Stench Blossom, generally agreement levels between tools, between tools and humans, and even between humans were poor. Our results on the performance of detection tools are in keeping with those of Fontana et al. [2011]. Our results on the performance of human inspectors call into question Marinescu [2004]'s study reporting that the accuracy for the manual detection of smells is about 87%. We found poor and variable agreement rates between inspectors. But detection will be influenced by the number, quality and preparation of the inspectors.

The general poor detection performance of tools and humans is exacerbated by the problem of having no baseline smell data with which to compare detection performance. It is impossible to know for real systems, like those in this study, which detection approach is actually best. We believe that the detection performance problems explicitly revealed by our study are endemic (though not made explicit) in other studies which detect smells. Our evaluations reveal a significant challenge for the smell community to address for the future.

However, the important thing about our evaluation results is that although tools and people detect different code constructs as containing a smell, the definition of those constructs must be explicit and transparent. As long as the construct that is being defined as a specific smell is transparent and accessible (as ours are), future researchers can understand, evolve, and converge on a more widespread understanding of what a smell actually is and the smell nuances that are important.

Table XVI. Summary of Research Data

| Projects | Packages | Releases | Sizes (KLOC) | Sizes (# files) | Maturities (number of previous releases) |
|---|---|---|---|---|---|
| Eclipse | JDT Core | 3.1 | 154 | 560 | 11 |
| Apache Commons | Common Codec | 1.3 | 5 | 22 | 3 |
| | Common DBCP | 1.2.1 | 10 | 25 | 3 |
| | Common DbUtils | 1.1 | 3 | 19 | 1 |
| | Common IO | 1.3.2 | 12 | 47 | 5 |
| | Common Logging | 1.1 | 7 | 10 | 6 |
| | Common Net | 1.4.1 | 33 | 68 | 7 |
| ArgoUML | All | 0.26 Beta 1 | 274 | 1,582 | 11 |

### 3.6. Open-Source Systems Used in this Investigation

We use source code and repository data from Eclipse, ArgoUML, and Apache Commons. We chose these three projects because they are well-known, mature systems that have a significant amount of change and fault data in their repositories. All three systems are written in Java. Table XVI summarises the releases and packages of the systems that we use. Table XVI shows that the six Apache Commons projects that we selected cover a variety of application areas (including database handling and networking). A rich variety of faults are likely to occur in these different projects.

Overall, we identified 560 source code files from release 3.1 of the core package of the Eclipse project, 1,582 source code files from the whole of ArgoUML, and 191 source code files from the six Apache Commons projects. JUnit tests were excluded from our datasets. The distribution of data is available, respectively, in Figure 1, Figure 2, and Figure 3 of Appendix C.

### 3.7. Fault Data Collection

There is no completely accurate way in which to collect fault data from open-source projects [Illes-Seifert and Paech 2008; Zeller 2013]. Our previous study [Hall et al. 2010] comparing three approaches to fault data collection suggests that the precision and recall of the Zimmermann et al. [2007] approach to identifying software faults from open-source projects is not as competitive as a manual approach. However, Zimmermann et al.'s approach is more practical than a manual approach for large datasets and is commonly used in studies of faults (e.g., [Śliwerski et al. 2005; Schröter et al. 2006; Illes-Seifert and Paech 2008; Nagappan et al. 2006]). Zimmermann et al.'s approach is so commonly used that it could be argued it has become the de-facto fault data collection approach. Consequently, we used Zimmermann et al.'s approach to collect fault data. Zimmermann et al.'s approach identifies software faults by locating bug fixes in the version configuration repository and then confirms these using Bugzilla bug reports. Zimmermann et al.'s approach is summarised as follows.

(1) Locate 'bug', 'fix(ed)' and 'update(d)' tokens in check-in comments.
(2) If a version entry contains one or more tokens and those tokens are followed by numbers, this version entry is seen as a potential bug fix.
(3) Those numbers are treated as bug IDs.
(4) The bug IDs are checked with bug reports from the bug reporting system (e.g., Bugzilla, JIRA). If a bug ID can be found in a bug report, the corresponding version entry can be seen as a bug fix.

We developed an Apache Ant script to automate this process of collecting fault data. Apache Ant is a Java-based script engine. We chose to use Apache Ant as it includes both CVS and SVN plugins which meant we could write scripts to download source code automatically. We applied our Apache Ant script to collect faults from two[14] releases of each project using our fault identification script. We then associated each fault with each source-code file. We calculated the number of faults related to a particular source code release as the number of identified fault fixes between this release and the next release. We also considered a fault fix of a fault fix (i.e., where a fix is made to a previously fixed piece of code) as two faults rather than a single fault. Previous studies show that a fix is likely to introduce new problems [Śliwerski et al. 2005; Kim et al. 2007], so ignoring fixes of a fix could miss useful data. Our approach means that each file has a count of the number of faults found in that file (i.e., it is a continuous or integer variable). Although we could have collected fault data at a lower level of granularity, for example, we could have identified the line(s) of code that contained the fault and established whether those faulty LOC contained a smell, we did not do this for two reasons. First, Zimmermann et al.'s approach reports at the file level. Second, smelly code does not necessarily have a direct impact on faults. It is likely that smelly code in a file produces code that is structured in such a way as to make inserting faults more likely. Consequently, we wanted to examine the relationship between faults and smells at a higher level of granularity than the line of code.

All of our raw fault and smell data is available to other researchers at `https://bugcatcher.stca.herts.ac.uk/badsmells`.

### 3.8. Statistical Analysis Technique

Many statistical analysis techniques require data to be normally distributed. Before identifying an appropriate analysis technique, we tested the distribution of our data. Figure 4 (Appendix E) shows that our dependent variable, that is, the number of faults in files, is not normally distributed across any of the three systems. This distribution is to be expected given that the vast majority of files are likely to contain no faults. We then tested whether the fault data was more normal if, rather than using fault counts, we used fault densities (i.e., the number of faults per KLOC in a file). Fault density normalises for the likelihood that larger files will contain more faults. Figure 5 (Appendix E) shows that fault density is not normally distributed in any of the systems. This lack of normally-distributed data means that many techniques should not be applied to this data (e.g., ANOVA is not an appropriate technique for our data).

The number of faults is a count variable, as it is a randomly generated nonnegative integer. The Poisson distribution is the simplest way to describe the distribution of such a variable. A key feature of the Poisson distribution is that its mean and variance are equal [Dobson 2010]. In practice, real data usually have variability exceeding the expected value by the Poisson. This phenomenon is called *over-dispersion*. Therefore, when the Poisson is too simple to model the count data, the negative binomial distribution is usually applied because it permits the variance to exceed the mean.

We applied a test for over-dispersion in our data. The null hypothesis is that the variance of the data is equal to the expected mean. The results in Table XXV (Appendix A) show that there is very strong evidence ($p - value < 0.001$) against the null hypothesis over all datasets. We therefore use negative binomial regression [Coxe et al. 2009] on the datasets. This technique is also suitable for the combination of continuous and

[14]Zimmermann et al. [2007] approach identifies faults by analysing check-in comment messages. To collect faults from a particular release, check-in comments from the next release must also be collected. Consequently, we collected faults from two releases of each project but only identified smells using one release - the first release.

categorical data that we will be using. The formula for negative binomial regression is

$$ln(dv) = intercept + c_1 p_1 + c_2 p_2 \ldots c_n p_n, \quad (1)$$

$$dv = e^{intercept + c_1 p_1 + c_2 p_2 \ldots c_n p_n}. \quad (2)$$

We also employed the *generalised likelihood ratio test* (GLRT) [Dobson 2010] to decide whether a complex negative binomial regression model should be used or not. GLRT says that when the simpler model is correct, twice the difference between the maximised value of the log-likelihood from the simpler model and that for the more complex one is approximately chi-square distributed.

In our review of 208 previous fault prediction studies [Hall et al. 2012], we found that several other research groups have used negative binomial regression in their studies of software faults (e.g., [Ostrand et al. 2005; Succi et al. 2003; Gao and Khoshgoftaar 2007]).

Finally, we assessed the multicollinearity among variables using variation inflation factor (VIF). Tables XXVI, XXVII, and XXVIII in Appendix B shows VIF values for each dataset. It can be seen that all VIF values are less than 2. Note that a VIF >10 is a sign of multicollinearity, suggesting either there are high correlations between pairs of variables, or there would be a significant change in the regression estimates after a minor change in the data.

## 4. RESULTS

### 4.1. Descriptive Statistics

Full raw data (both smell and fault data) are available from our online supplementary material[15] with the scatter plot matrices in Appendix C providing further information on the distribution of data in each of the three systems. These plots show the distribution between any two of the variables that we are studying and allow for visual inspection of co-located features. Appendix C shows that the density and distribution of smells vary in each system. Tables XXIX, XXX, and XXXI in Appendix D shows the correlations between independent variables. Most of the correlations are low (<0.30)[16], apart from in Eclipse where there is a strong correlation between LOC and Message Chains, as well as a weak correlation between LOC and Switch Statements. Correlation between smells and LOC has been previously reported (as discussed in Section 2.2), and this is the reason we included LOC in our study. Correlation between smells has also been previously reported (as discussed in Section 2.1). However, we found no correlation between the smells we investigated. The lack of correlation demonstrates that these smells are almost orthogonal features and further justifies our choice of the bad smells.

### 4.2. Relationships between Code Smells and Faults

We applied negative binomial regression to Eclipse, Apache Commons, and ArgoUML datasets. Our negative binomial regression models are built with number of faults in a file as the dependent variable and number of LOC, together with Data Clumps (existence of), Middle Man (existence of), Switch Statements (number of), Speculative Generality (existence of), and Message Chains (number of) as the independent variables. All these variables are at the file level.

For each dataset, we iteratively built an interaction model that best fits the data. The first-order interaction model involves all independent variables. Our first-order interaction model does two things: first, it separates the impact of each independent

[15] https://bugcatcher.stca.herts.ac.uk/tosem2014/.

[16] Correlations above 0.5 are strong.

Table XVII. First-Order Interaction Negative Binomial Regression Model on the Eclipse Data

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| **(Intercept)** | −0.2118 | 0.0849 | −2.50 | **0.0126** |
| **LOC** | 0.0023 | 0.0005 | 4.87 | **0.0000** |
| **DataClumps** | −0.6626 | 0.2153 | −3.08 | **0.0021** |
| MiddleMan | −0.3143 | 0.1944 | −1.62 | 0.1059 |
| **SpeculativeGenerality** | −0.1096 | 0.0499 | −2.20 | **0.0280** |
| SwitchStatements | −0.3686 | 0.3111 | −1.19 | 0.2360 |
| **MessageChains** | 0.0832 | 0.0332 | 2.51 | **0.0122** |
| **LOC:DataClumps** | 0.0025 | 0.0009 | 2.87 | **0.0041** |
| LOC:MiddleMan | −0.0014 | 0.0010 | −1.42 | 0.1570 |
| **LOC:SpeculativeGenerality** | 0.0005 | 0.0002 | 2.75 | **0.0059** |
| LOC:SwitchStatements | −0.0009 | 0.0008 | −1.10 | 0.2702 |
| **LOC:MessageChains** | −0.0002 | 0.0001 | −2.64 | **0.0083** |
| DataClumps:MiddleMan | 0.2893 | 0.3966 | 0.73 | 0.4657 |
| DataClumps:SpeculativeGenerality | −0.0354 | 0.0896 | −0.39 | 0.6929 |
| DataClumps:SwitchStatements | −0.4971 | 0.5102 | −0.97 | 0.3299 |
| DataClumps:MessageChains | 0.0176 | 0.0331 | 0.53 | 0.5947 |
| MiddleMan:SpeculativeGenerality | 0.0199 | 0.1027 | 0.19 | 0.8467 |
| **MiddleMan:SwitchStatements** | 0.8255 | 0.4189 | 1.97 | **0.0488** |
| MiddleMan:MessageChains | 0.0152 | 0.0312 | 0.49 | 0.6255 |
| SpeculativeGenerality:SwitchStatements | −0.1714 | 0.1318 | −1.30 | 0.1934 |
| SpeculativeGenerality:MessageChains | −0.0160 | 0.0098 | −1.63 | 0.1023 |
| **SwitchStatements:MessageChains** | 0.1815 | 0.0574 | 3.16 | **0.0016** |

variable on faults from the impact of all other independent variables; second, it analyses the impact of every pair of independent variables on faults.[17] We select from this first order model only the terms significantly associated with faults and build a simpler model. We continue iteratively building simpler and better fitted models until we have built the simplest model for the data. We explain in detail this iterative model-building process in the next sub-section where we build the model for the Eclipse dataset.

*4.2.1. Eclipse.* Table XVII shows the first-order interaction model for Eclipse. Each row in the table represents the smell tested in relation to the number of faults in a file. Combinations of smells are also tested using multiplicative interaction terms, that is, when smells occur in a file that we are measuring as categorical or binary variables, then the interaction term evaluates to 1 ($1 \times 1$) if both smells are present in a file. For example, the DataClumps:MiddleMan row tests for a relationship between faults and occurrences of both the Data Clumps and Middle Man smell in files if both these smells are present in a file the interaction term evaluates to 1. These interaction terms ignore the presence or absence of any other smell in the file. Testing the relationship between LOC and faults is also included in the table. The Intercept row in the table summarises the average impact on faults of factors not explicitly mentioned in the table.

The first-order interaction model for Eclipse (shown in Table XVII) has a residual deviance equal to 531.1 on 538 degrees of freedom (DF). Because its residual deviance is no larger than its DF, the model fits acceptably. However, Table XVII shows that LOC, DataClumps, SpeculativeGenerality, MessageChains, LOC:DataClumps, LOC:SpeculativeGenerality, LOC:MessageChains, MiddleMan:SwitchStatements, and SwitchStatements:MessageChains are the only terms significantly ($p - value < 0.05$)

[17]We did not build an interaction model based on all combinations of our independent variables, as there was not enough data for this analysis.

Table XVIII. Simpler Negative Binomial Regression Model on the Eclipse Data

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| **(Intercept)** | −0.2581 | 0.0798 | −3.24 | **0.0012** |
| **LOC** | 0.0021 | 0.0004 | 4.72 | **0.0000** |
| **DataClumps** | −0.5845 | 0.2051 | −2.85 | **0.0044** |
| **SpeculativeGenerality** | −0.1220 | 0.0465 | −2.62 | **0.0087** |
| **MessageChains** | 0.0850 | 0.0225 | 3.78 | **0.0002** |
| **LOC:DataClumps** | 0.0020 | 0.0006 | 3.50 | **0.0005** |
| LOC:SpeculativeGenerality | 0.0002 | 0.0001 | 1.41 | 0.1578 |
| **LOC:MessageChains** | −0.0002 | 0.0000 | −3.28 | **0.0010** |
| MiddleMan:SwitchStatements | −0.0487 | 0.2580 | −0.19 | 0.8503 |
| MessageChains:SwitchStatements | 0.0268 | 0.0329 | 0.82 | 0.4144 |

Table XIX. Much Simpler Negative Binomial Regression Model on the Eclipse Data

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| **(Intercept)** | −0.2956 | 0.0762 | −3.88 | **0.0001** |
| **LOC** | 0.0024 | 0.0004 | 6.28 | **0.0000** |
| **DataClumps** | −0.5987 | 0.2058 | −2.91 | **0.0036** |
| **SpeculativeGenerality** | −0.0811 | 0.0317 | −2.56 | **0.0104** |
| **MessageChains** | 0.0667 | 0.0203 | 3.28 | **0.0010** |
| **LOC:DataClumps** | 0.0021 | 0.0006 | 3.52 | **0.0004** |
| **LOC:MessageChains** | −0.0001 | 0.0000 | −2.93 | **0.0034** |

related to numbers of faults. Therefore, we develop a simpler model using only these terms.

Table XVIII shows this simpler model for Eclipse. The simpler model is more acceptable than the first-order model because its residual deviance 531.4 is less than 550 degrees of freedom. Furthermore, in the simpler model LOC, DataClumps, SpeculativeGenerality, MessageChains, LOC:DataClumps, and LOC:MessageChains are significant ($p - value < 0.05$).

We now use these six significant terms to fit a much simpler model, whose results are shown in Table XIX. The model has a residual deviance 527.8 on 553 degrees of freedom. We can test the overall effect of the much simpler model by comparing the log likelihood of the first-order model with the model excluding those six terms using GLRT. The change in log-likelihood is about 28.25, which is larger than the change of 9 in DF, suggesting those six terms are most significant to the fitted model.

The estimates of these three models change, for example, Intercept is about −0.2118 in the first model (Table XVII), then −0.2581 in the simpler model (Table XVIII), and −0.2956 in the final model (Table XIX). The reason for this change is that the meaning of these Estimate Values depends on what other independent variables are in each model. For example, Intercept in the first model can be interpreted as the expectation of log value of the number of faults when the number of LOC, Data Clumps, Middle Man, Switch Statements, Speculative Generality, and Message Chains are all equal to 0. In the final model, Intercept is the expectation of log value of the number of faults when only the number of LOC and Message Chains are equal to 0. So, Intercept must be the best estimate of log value of the number of faults over the ranges of values of Data Clumps, Middle Man, Switch Statements, and Speculative Generality in the final model.

In Table XIX, LOC, MessageChains, and LOC:DataClumps are associated with higher numbers of faults. SpeculativeGenerality, LOC:DataClumps, and

Table XX. Negative Binomial Regression Model on the ArgoUml Data

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| **(Intercept)** | −2.4080 | 0.1200 | −20.06 | **0.0000** |
| **LOC** | 0.0045 | 0.0009 | 4.97 | **0.0000** |
| **DataClumps** | 1.1684 | 0.3822 | 3.06 | **0.0022** |
| **MiddleMan** | −0.6398 | 0.2683 | −2.39 | **0.0171** |
| **SpeculativeGenerality** | 0.1336 | 0.0616 | 2.17 | **0.0301** |
| SwitchStatements | 0.4956 | 0.6148 | 0.81 | 0.4202 |
| **MessageChains** | 0.0676 | 0.0208 | 3.25 | **0.0011** |
| **LOC:DataClumps** | −0.0049 | 0.0011 | −4.63 | **0.0000** |
| LOC:MiddleMan | 0.0008 | 0.0013 | 0.58 | 0.5599 |
| **LOC:SpeculativeGenerality** | 0.0003 | 0.0002 | 2.09 | **0.0370** |
| LOC:SwitchStatements | −0.0001 | 0.0009 | −0.10 | 0.9184 |
| **LOC:MessageChains** | −0.0001 | 0.0000 | −4.00 | **0.0001** |
| DataClumps:MiddleMan | 0.8852 | 0.5649 | 1.57 | 0.1171 |
| DataClumps:SpeculativeGenerality | −0.1137 | 0.0884 | −1.29 | 0.1986 |
| DataClumps:SwitchStatements | 1.0369 | 0.7532 | 1.38 | 0.1686 |
| DataClumps:MessageChains | 0.0445 | 0.0325 | 1.37 | 0.1710 |
| **MiddleMan:SpeculativeGenerality** | −0.2723 | 0.1049 | −2.60 | **0.0094** |
| MiddleMan:SwitchStatements | 0.3184 | 1.0158 | 0.31 | 0.7540 |
| **MiddleMan:MessageChains** | 0.0953 | 0.0306 | 3.11 | **0.0018** |
| SpeculativeGenerality:SwitchStatements | −0.2823 | 0.2141 | −1.32 | 0.1873 |
| SpeculativeGenerality:MessageChains | −0.0084 | 0.0047 | −1.80 | 0.0711 |
| SwitchStatements:MessageChains | −0.1069 | 0.0817 | −1.31 | 0.1911 |

LOC:MessageChains are negatively related to the number of faults.[18] This negative relation means that the occurrence of, for example, LOC:MessageChains is related to lower number of faults. Figure 6 (Appendix F) visualises faults in relation to LOC and Message Chains from the Eclipse model in Table XIX and shows that the model fits the data well. Figure 6 shows that Eclipse files with higher lines of code tend, as expected, to have higher numbers of faults, and files with higher numbers of Message Chains tend to have higher numbers of faults. However, files with higher lines of code and higher numbers of Message Chains tend to have lower numbers of faults. For brevity, we include only one visualisation of the Eclipse data.

Despite the significant associations shown in Table XIX, the effect of these associations on the numbers of faults is small. The McFadden effect size statistic [McFadden 1977] for Table XIX is 0.08. This effect size is small, showing that only 8 percent of the faults in Eclipse can be accounted for by the variables in Table XIX. The significant Intercept in Table XIX confirms that there are other factors significantly affecting faults that do not appear in the model.

*4.2.2. ArgoUML.* A similar analysis has been carried out on the ArgoUML dataset. Table XX shows the first-order model built for ArgoUML. The model has a residual deviance equal to 655.4 on 1,560 DF. A simpler model is built by iteratively removing nonsignificant terms, and the results are shown in Table XXI. All terms in this simpler model are highly significant. This model has a residual deviance equal to 641.2 on 1,575 DF. Table XXI shows that ArgoUML files with high numbers of either Data Clumps or Message Chains or files with high numbers of lines of code are faulty, but files with both high LOC and Data Clumps or Message Chains are less faulty. Table XXI also shows

[18]A factor that reduces the number of faults is indicated by a negative estimated coefficients (that is values in the Estimate column in the table) and visa versa.

Table XXI. Simpler Negative Binomial Regression Model on the ArgoUml Data

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| **(Intercept)** | −2.3600 | 0.1117 | −21.13 | **0.0000** |
| **LOC** | 0.0049 | 0.0007 | 7.14 | **0.0000** |
| **DataClumps** | 1.7781 | 0.2554 | 6.96 | **0.0000** |
| **MiddleMan** | −0.5163 | 0.2007 | −2.57 | **0.0101** |
| **MessageChains** | 0.0620 | 0.0174 | 3.56 | **0.0004** |
| **LOC:DataClumps** | −0.0047 | 0.0007 | −6.47 | **0.0000** |
| **LOC:MessageChains** | −0.0001 | 0.0000 | −3.91 | **0.0001** |

Table XXII. Negative Binomial Regression Model on the Apache Data

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| (Intercept) | 0.3167 | 0.2355 | 1.34 | 0.1787 |
| **LOC** | 0.0032 | 0.0013 | 2.48 | **0.0130** |
| **DataClumps** | −6.9503 | 2.0258 | −3.43 | **0.0006** |
| MiddleMan | −0.1794 | 0.3315 | −0.54 | 0.5884 |
| SpeculativeGenerality | −0.2046 | 0.1906 | −1.07 | 0.2832 |
| SwitchStatements | −1.5977 | 0.9429 | −1.69 | 0.0902 |
| **LOC:DataClumps** | 0.0089 | 0.0033 | 2.71 | **0.0068** |
| LOC:MiddleMan | −0.0002 | 0.0025 | −0.09 | 0.9278 |
| LOC:SpeculativeGenerality | −0.0030 | 0.0016 | −1.83 | 0.0670 |
| LOC:SwitchStatements | 0.0039 | 0.0026 | 1.49 | 0.1351 |
| DataClumps:MiddleMan | 2.7708 | 1.6662 | 1.66 | 0.0963 |
| **DataClumps:SpeculativeGenerality** | 1.4652 | 0.4775 | 3.07 | **0.0022** |
| MiddleMan:SpeculativeGenerality | −0.2482 | 0.6232 | −0.40 | 0.6904 |
| SpeculativeGenerality:SwitchStatements | −1.0080 | 1.9404 | −0.52 | 0.6034 |

that Middle Man reduces the number of faults in ArgoUML. Figure 7 (Appendix F) visualises faults in relation to LOC and Message Chains from the ArgoUML model in Table XXI and shows that the model fits the data well. For brevity, we include only one visualisation of the ArgoUML data.

A similar effect size can be measured for ArgoUML as has also been done for the Eclipse dataset. The McFadden effect size statistic for Table XXI is 0.08. This is again a small effect size showing that only 8 percent of faults can be accounted for by the variables in Table XXI. The significant Intercept in Table XXI, again, confirms that there are other factors significantly affecting faults that do not appear in the model.

*4.2.3. Apache Commons.* A similar analysis has been carried out on the Apache Commons dataset. Table XXII shows the first-order model built for Apache Commons. No Message Chains appear in the table, because only three files contained Message Chains. Consequently we deleted this variable, as there was not enough data to build a valid model. The model we did build has a residual deviance equal to 147.5 on 177 DF. A simpler model is iteratively built, and the results are shown in Table XXIII. DataClumps and DataClumps:LOC remain significant in Table XXIII. This model has a residual deviance equal to 154.9 on 188 DF. Figure 8 (Appendix F) visualises this model and shows that the data fits the model well.

A similar effect size can be measured for Apache as has been done for the Eclipse and ArgoUML datasets. The McFadden effect size statistic for Table XXIII is 0.07. This is also a small effect size showing that only 7 percent of faults can be accounted for by

Table XXIII. Simpler Negative Binomial Regression Model on the Apache Data

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| **(Intercept)** | 0.3471 | 0.1296 | 2.68 | **0.0074** |
| **DataClumps** | −4.0870 | 0.8374 | −4.88 | **0.0000** |
| **DataClumps:LOC** | 0.0080 | 0.0019 | 4.11 | **0.0000** |

the variables in Table XXIII. The significant Intercept in XXIII, again, confirms that there are other factors significantly affecting faults that do not appear in the model.

*4.2.4. Summary of Results.* Table XXIV provides an overview of our results for the three systems. There is no significant relationship in any of the three systems between faults and Switch Statements. There are significant relationships between faults and LOC and Message Chains across two systems. Where there is a relationship between faults and smells, it is not always a positive relationship. Some smells (e.g., Middle Man in ArgoUML) are associated with fewer rather than more faults. Table XXIV suggests that no consistent relationships exist between particular smells and faults across all three systems. Commonality does exist between Eclipse and ArgoUML and between Eclipse and Apache Commons in the relationships between some smells and faults. The significant relationships shown in Table XXIV all have relatively small effect sizes, which means that where smells seem to affect fault numbers, the impact is small. Factors other than smells have a greater effect on faults. The results presented allow us to respond to our initial hypotheses as follows.

*Hypotheses* 1. The Data Clumps Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.
Reject. Data Clumps seem to effect all three of the systems. Data Clumps is related to fewer faults in two of the systems and related to more faults in one system. Where Data Clumps are combined with larger files, more faults can be observed in two systems and fewer faults in one system. The effect of Data Clumps on the number of faults is small.

*Hypotheses* 2. The Middle Man Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.
Partially reject. Middle Man Smells are related to fewer faults in some but not all systems. Where Middle Man Smells affect faults the size of that effect is small.

*Hypotheses* 3. The Speculative Generality Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.
Partially reject. Speculative Generality Smells are related to fewer faults in one system. Where Speculative Generality Smells affect faults the size of that effect is small.

*Hypotheses* 4. The Switch Statements Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.
Accept. The Switch Statements Smell seems not to affect faults in the systems.

*Hypotheses* 5. The Message Chain Smell has no effect (either on its own or in combination with other smells) on numbers of faults in files.
Partially reject. In the two systems that contained Message Chains, they are related to more faults. Where the Message Chain Smell is combined with larger files, fewer faults can be observed in these two systems. Where the Message Chain Smell affects faults, the size of that effect is small.

Table XXIV. Summary of the Impact of Smells and LOC on Faults

| | Eclipse | ArgoUML | Apache Commons |
|---|---|---|---|
| LOC | positive | positive | |
| DataClumps | negative | positive | negative |
| MiddleMan | | negative | |
| SpeculativeGenerality | negative | | |
| SwitchStatements | | | |
| MessageChains | positive | positive | |
| LOC:DataClumps | positive | negative | positive |
| LOC:MiddleMan | | | |
| LOC:SpeculativeGenerality | | | |
| LOC:SwitchStatements | | | |
| LOC:MessageChains | negative | negative | |
| DataClumps:MiddleMan | | | |
| DataClumps:SpeculativeGenerality | | | |
| DataClumps:SwitchStatements | | | |
| DataClumps:MessageChains | | | |
| MiddleMan:SpeculativeGenerality | | | |
| MiddleMan:SwitchStatements | | | |
| MiddleMan:MessageChains | | | |
| SpeculativeGenerality:SwitchStatements | | | |
| SpeculativeGenerality:MessageChains | | | |
| SwitchStatements:MessageChains | | | |

positive = associated with more faults; negative = associated with fewer faults

*Hypothesis* 6. The size of files is not related to numbers of faults in files.

Partially reject. The size of files seems to be related to more faults in some but not all systems. Where the size of files is combined with the Message Chain Smell, fewer faults can be observed. Where the size of files affects faults, the size of that effect is small.

## 5. DISCUSSION

Our findings contribute important evidence for future researchers and practitioners on the effect of smells on faults.

### 5.1. Some Smells are not Related to Faults

In the three systems that we analysed, we found no evidence of a relationship between faults and Switch Statements. This is an important finding, as it highlights for researchers a smell likely to be unimportant in future studies of faults. This finding also provides practitioners with valuable information on refactoring: refactoring Switch Statements is unlikely to reduce faults in their systems. It is important that results are published which identify smells where no effect on faults can be found. Such results allow practitioners and future researchers to direct their effort towards identifying and prioritising refactoring smells most likely to actually be related to faults. Our findings add Switch Statements to the catalogue of smells identified by previous studies (e.g., [Li and Shatnawi 2007; D'Ambros et al. 2010]) as having no effect on faults. Our results also add to the growing evidence that many smells may not be worth refactoring. For example, Sjøberg et al. [2013] report that the 12 smells that they investigated had no significant effect on increasing maintenance effort in the four systems that they studied.

### 5.2. Smell Effects are Unstable across Systems

None of the five smells that we studied have a consistent effect on faults across all three systems. Some commonality exists between Eclipse and the other two systems. Many previous studies report that fault prediction models perform very poorly when transferred to other systems [Bell et al. 2006; Denaro and Pezzè 2002; Nagappan et al. 2006]. The differing results that we present may explain some of this fault prediction model performance instability. The same smell (or independent variable) has different effects on faults across different systems, because smells manifest differently in different systems. Systems are developed for different applications in different environments by different developers often using different coding styles. Consequently, the code produced will have different characteristics. For example, in the three systems that we studied, Apache Commons developers used very few Message Chains. Consequently it is not surprising that despite Message Chains being significantly related to faults in Eclipse and ArgoUML, they were not related to faults in Apache Commons. It is important that future work defines the different manifestations of the same smell and identifies those manifestations that affect faults.

The inconsistent findings reported by previous smell studies may be partially explained in terms of the different systems that have been used in these studies. For example, Duplicate Code is reported by some studies to increase faults [Juergens et al. 2009] but by other studies to reduce faults [Rahman et al. 2010]. The characteristics of the smells in the individual systems must be accounted for in future studies.

### 5.3. Some Smells Increase Faults and Other Smells Reduce Faults

Some smells do have statistically significant relationships with faults in some systems. The occurrence of some smells is related to more faults, but the occurrence of other smells is related to fewer faults.

Message Chains are related to higher numbers of faults in Eclipse and ArgoUML files. This finding may well demonstrate the dangers of breaking the Law of Demeter [Lieberherr et al. 1988], which warns against using objects to access subsequent objects. We add Message Chains to the catalogue of smells reported previously (e.g., Shotgun Surgery, God Class, and God Method [Li and Shatnawi 2007]) to increase the number of faults in some circumstances in some systems.

Middle Man smells are related to fewer faults in ArgoUML, and Data Clumps are related to fewer faults in Eclipse and Apache Commons. Currently, it is difficult to explain why this might be the case. However, our results enable us to add Data Clumps, Speculative Generality, and Middle Man to the catalogue of smells reported previously as decreasing the number of faults in some circumstances in some systems (e.g., Duplicated Code [Rahman et al. 2010]).

### 5.4. Code Size Is an Important Consideration

Our results suggest that code size alone can affect faults; increased lines of code related to increased faults in both Eclipse and ArgoUML files. Code size alone did not affect faults in Apache Commons files, however, the size of files in Apache Commons tends to be smaller than files in the other two systems.

Our results also suggest that increased file size may mitigate the effect that Message Chains have on increased faults in Eclipse and ArgoUML. Our results showed increased faults where Message Chains occur in Eclipse and ArgoUML files, but where Message Chains occur together with increased file size, we found fewer faults. A manual inspection of larger-than-average files with many Message Chains suggests that such files are often God Classes. God Classes have previously been identified as having fewer faults [Olbrich et al. 2010]. Perhaps because God Classes are frequently used,

they may be more thoroughly tested and any faults are identified and fixed early in the lifecycle. We also manually inspected smaller-than-average files with many Message Chains (there are few of these files). Such files tended to also have few faults. We found that these files are usually 'views' from the model-view-controller design pattern. It is possible that such files are relatively easy to visually inspect and so their low faultiness is not surprising. There are many more small files without Message Chains which have few faults. However, in ArgoUML, there are 14 files which are smaller than average without Message Chains, but which are faulty. We found that ten of these files were 'models' from the model-view-controller pattern. This finding suggests that while one element of this pattern (the view) is less fault-prone, another element (the model) is fault-prone. It may be that while the view element is easy to visually check for faults, the model element is less easy to visually check for faults. Investigating the relationship between this design pattern and faults needs to be investigated further in the future. We performed a similar manual inspection of files in relation to the occurrence of Data Clumps. However, we could find no discernible explanation for the faults found in either small or large files containing Data Clumps.

Overall, our results suggest that the impact of code size on faults is not clear. Despite the extensive body of previous work on code size, its effect on faults remains poorly understood. More work is still needed to unravel the relationship between size and the number of faults, especially in the context of smells. Our results suggest the effect which size has on smells and their relationship with faults may not be as straightforward as Vokac [2004] and Olbrich et al. [2010] report.

### 5.5. The Effect Size of Smells on Faults Is Small

Our results show that where smells do affect faults, the size of that effect is relatively small. Yamashita and Moonen [2013b] also report that smells have only a minor effect on maintenance problems. In their study, only 30% of files that gave problems during maintenance contained any of the 12 smells they investigated. All of the effects of smells that we report are under 10 percent. Other factors have much more effect on faults than the smells we consider. This is a very important finding and suggests that fault reduction strategies which focus only on refactoring smells (and on file size) will have limited impact. Previous studies have reported on various other factors that affect faults. For example, a combination of socio-technical factors have been reported to affect faults [Bird et al. 2009], and also a combination of static code metrics, process metrics, and source-code text demonstrates good fault detection [Shivaji et al. 2009]. It is likely that a range of factors need to included in any approach to fault reduction.

It is critically important that studies report the effect size for any relationships reported between smells and faults. Most do not [Kampenes et al. 2007], a notable exception being Sjøberg et al. [2013]. Most simply report that a significant relationship exists between a smell and faults. Reporting only the significance of the relationship can give a misleading indication of the effect of the smell on faults. Our results show that even where smells do have a significant relationship with faults, the impact of smells on faults is minor. Practitioner and researcher effort is probably better focused on factors which have a larger effect on faults. It is likely that the number of faults is affected by very many factors. Future work should identify the sets of factors affecting faults and the size of their effects.

### 5.6. Smell Detection Methodology Is a Challenge

Our evaluation of smell detection performance shows that it is difficult to define and operationalise smell definitions, either for automatic or manual smell detection. Generally, agreement levels on what code contains a smell are poor between tools, between tools and humans, and even between humans. This general poor performance of tools

and humans is exacerbated by the problem of having no baseline smell data with which to compare detection performance (i.e., it is not possible to construct a definitive list of smells for a system against which to compare detections). So, it is impossible to know for real systems which detection approach is actually best. This problem is exacerbated by a lack of transparency in the smell definitions used by some tools. Smells are highly subjective and universally defining, and detecting smells is increasingly being recognised as challenging, if not impossible [Liu et al. 2013]. Our findings reveal significant, and so far largely ignored, methodological challenges for the smell community to address.

## 6. THREATS TO VALIDITY

### 6.1. Internal Validity

We did not fully evaluate the detection accuracy of our tool. In particular, we did not evaluate how accurately our tool detected Speculative Generality or Middle Man smells. However, we have no reason to believe that our tool performs any differently for these two smells than it does for those smells that we did evaluate. Furthermore, we have no evidence that our tool performs any worse than existing smell detection tools. Overall, it is difficult for any tool developer to evaluate detection performance on real systems, as definitive lists of smells do not exist. Calculating the precision of the tool is difficult.

We did not manually examine many files containing relatively high numbers of faults which also contained none of the five smells we investigate. Such an examination may have revealed other structures (and possibly other smells) that have a greater effect on faults and which may be confounding our results. We are planning a future study to investigate the effect size of a large number of factors on fault numbers.

### 6.2. External Validity

Only considering source code from open-source projects is a limitation. These systems may not be representative of the way developers develop systems more generally. As in all such studies that use open-source systems, our results may not be relevant to commercial software projects. However, the three systems that we do investigate cover a range of system sizes, application areas, and faults. In selecting these systems, it is likely that a wide variety of coding styles have been included, which generate a variety of smell and fault data.

One of the projects (Apache Commons) is made up of a collection of separate projects. The aggregation of the Apache Commons data may be a threat, because each project is independent and not necessarily written by the same developers. However, we do not believe this aggregation to be a problem, because first, more coding styles are introduced into the study, generating a wider range of smelly code to analyse; second, Eclipse and ArgoUML are also collections of packages and plug-ins performing very different tasks. All the individual items are usually used necessarily together for all three systems, for example, Apache libraries are almost always used together and not separately as individual systems.

### 6.3. Construct Validity

Smell definitions will always be subjective and moving from the original definitions, which are not formal and were probably never meant to be formal, to definitions that can be implemented in a tool will always be problematic. Widespread differences exist in the way that Fowler and Beck's smells have been interpreted and defined, both in the literature and in smell detection tools [Fontana et al. 2011]. As in DECOR [Moha et al. 2010], our smell definitions may be different from other researchers' and practitioners'

definitions. Our definitions are based on empirical evidence using a variety of experienced programmers. They are not perfect definitions, and not everyone will agree with them, but we are explicit about the code constructs that we are detecting, and this transparency will enable our definitions to be inspected, evaluated, and evolved by others.

Data Clumps could have been measured continuously, but that we decided to do so had a high risk of incorrect detections. This risk stems from (as Table II shows) the two types of Data Clumps: the first type is identified across classes and so a class will either contain or not contain a Data Clump; the second type is intra-class Data Clumps, where multiple methods have the same signature. The intra-class Data Clumps cannot always be precisely identified because the list of parameters in the method signature is not descriptive enough to uniquely identify specific Data Clumps. This is a threat to our analysis, yet categorical measurement is fairly standard in fault prediction studies where many previous studies even measure faults categorically (i.e., identifying only whether faults occur in a code unit or not, e.g., Shivaji et al. [2009]). However, categorical measurement may lose information about amounts of smells, that is, many Data Clumps in a file may be more important than only one. Similarly, the strength of the smell may also be important. For example, the length of a Message Chain may be important. Although measuring smell strength would be very powerful, doing so presents challenging definition issues and is beyond the scope of this investigation. As a consequence, our results may not be telling the full story in regards to the characteristics of smells in relation to faults. Investigating these issues are also part of a future study.

## 7. CONCLUSIONS

In this article, we investigated the effect of five smells on the number of faults in three systems. The only result that was consistent across all three systems was that Switch Statements had no effect on faults in any of the systems. This finding is important, as it suggests that when fault reduction is the focus of refactoring, spending effort on refactoring Switch Statements is unlikely to be effective.

Our other results varied across the three systems. For example, Data Clumps increased faults in one system, but reduced faults in the other two systems. Our results suggest that smells manifest differently in different systems, probably dependent on the application domain and development context. This is an important finding, as it shows that arbitrary refactoring across systems is unlikely to be effective. Future work is needed to establish the various manifestations of smells and identify those manifestations which affect faults.

We also found that some smells reduced the number of faults rather than increased them. For example, Middle Man reduced faults in ArgoUML, and Speculative Generality reduced faults in Eclipse. This is an important finding, as it suggests that the arbitrary refactoring of smells may actually be counterproductive.

We also investigated the effect of file size as a confounding factor to the relationship between smells and faults. Our results show that code size alone affects faults in some systems but not in all systems. This finding may explain why previous studies report conflicting findings on the effect of lines of code on faults. We also found that increased file size seems to mitigate the impact of Message Chains on increased faults. Where Message Chains occurred in larger files, their effect switched from increasing to reducing faults.

Even where we found that smells did significantly affect faults in files, the size of that effect was small (always under 10 percent). It seems that faults are affected by more factors than the five smells (and code size) that we investigated. It is essential that future work identify the set of factors that impact the number of faults in code and that those factors which have the most effect on faults are identified.

Finally, it is increasingly clear that there are many methodological variations in studies of smells. Most importantly, there are variations in the definitions of smells used by studies and in the operationalisation of those definitions in smell detection tools. These variations mean that different code is identified as containing the same smell. Such data collection inconsistency undermines the usefulness of individual studies, making it dangerous to compare the results of one study against another. These methodological inconsistencies pose a significant challenge for the smell community to address.

## APPENDIXES

## A. TEST RESULTS FOR OVER-DISPERSION

Table XXV. Test for Over-Dispersion on the Number of Defects with Different Datasets

| Data Set | Obs.Var/Theor.Var | Statistic | p-value |
|---|---|---|---|
| Apache | 4.46 | 846.77 | 0.00 |
| Eclipse | 3.40 | 1900.47 | 0.00 |
| ArgoUML | 2.03 | 3208.92 | 0.00 |

## B. SENSITIVITY OF MODELS

Table XXVI. Variance Inflation Factor Values for the Apache Dataset

| | Variance Inflation Factor |
|---|---|
| LOC | 1.12 |
| DataClumps | 1.08 |
| MiddleMan | 1.09 |
| SpeculativeGenerality | 1.03 |
| SwitchStatements | 1.10 |

Table XXVII. Variance Inflation Factor Values for the Eclipse Dataset

| | Variance Inflation Factor |
|---|---|
| LOC | 1.82 |
| DataClumps | 1.07 |
| MiddleMan | 1.10 |
| SpeculativeGenerality | 1.07 |
| SwitchStatements | 1.17 |
| MessageChains | 1.58 |

Table XXVIII. Variance Inflation Factor Values for the Apache Commons Dataset

| | Variance Inflation Factor |
|---|---|
| LOC | 1.20 |
| DataClumps | 1.16 |
| MiddleMan | 1.00 |
| SpeculativeGenerality | 1.12 |
| SwitchStatements | 1.05 |
| MessageChains | 1.11 |

## C. SCATTER PLOTS FOR DIFFERENT DATASETS

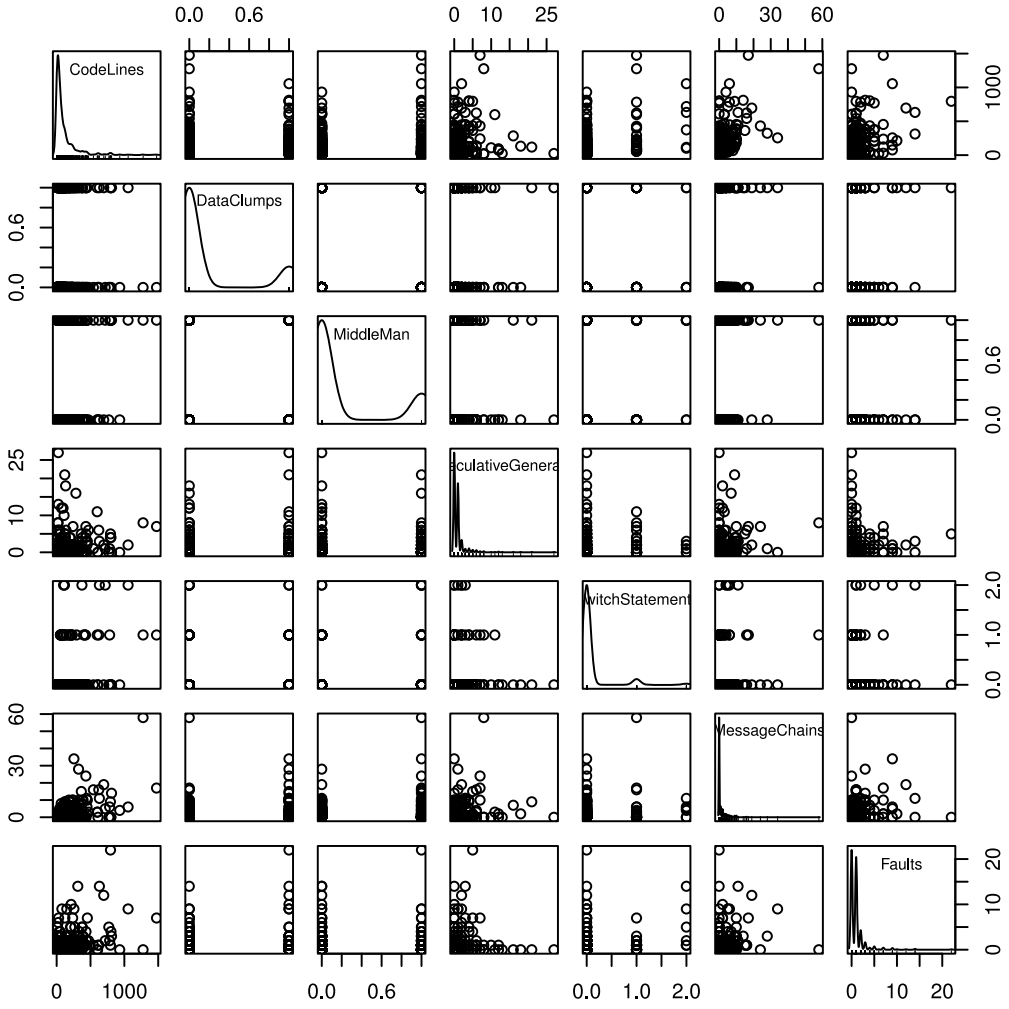


Fig. 1. A scatter plot matrix of Eclipse.

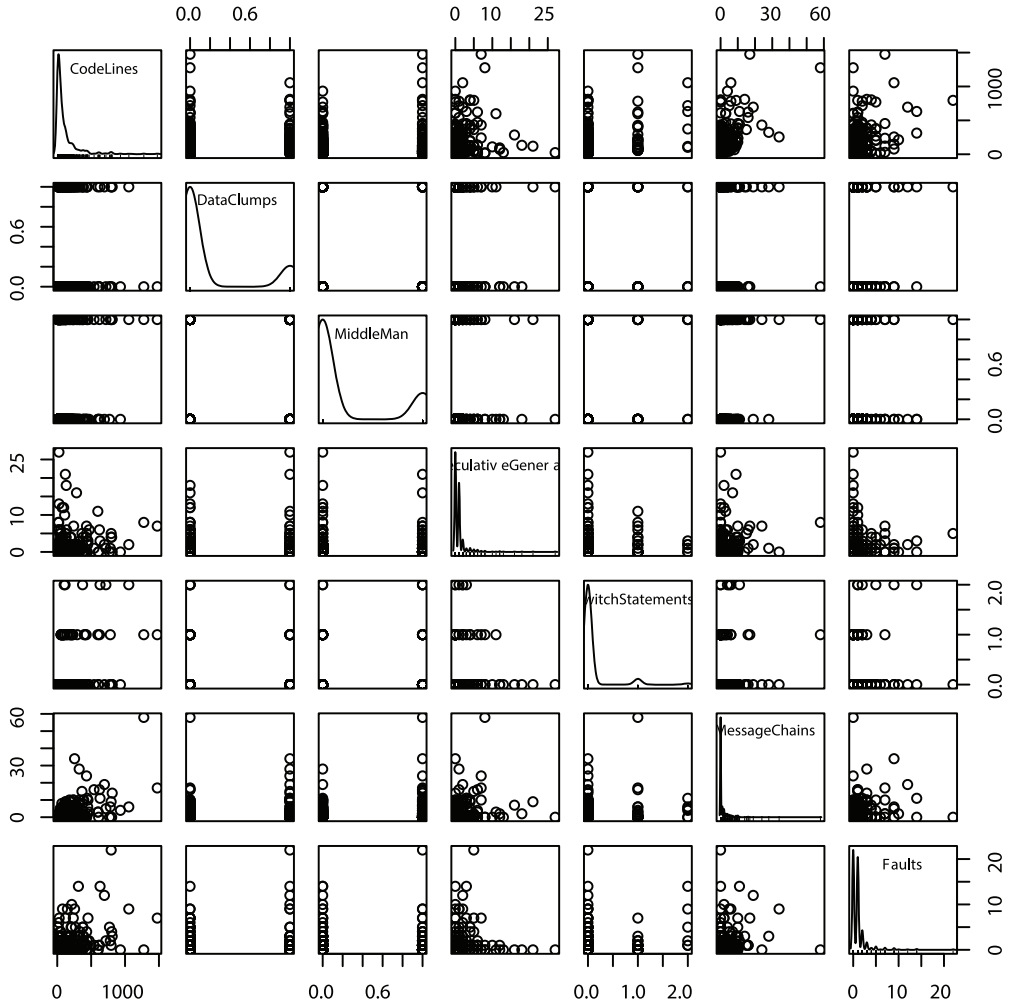


Fig. 2. A scatter plot matrix of ArgoUml.

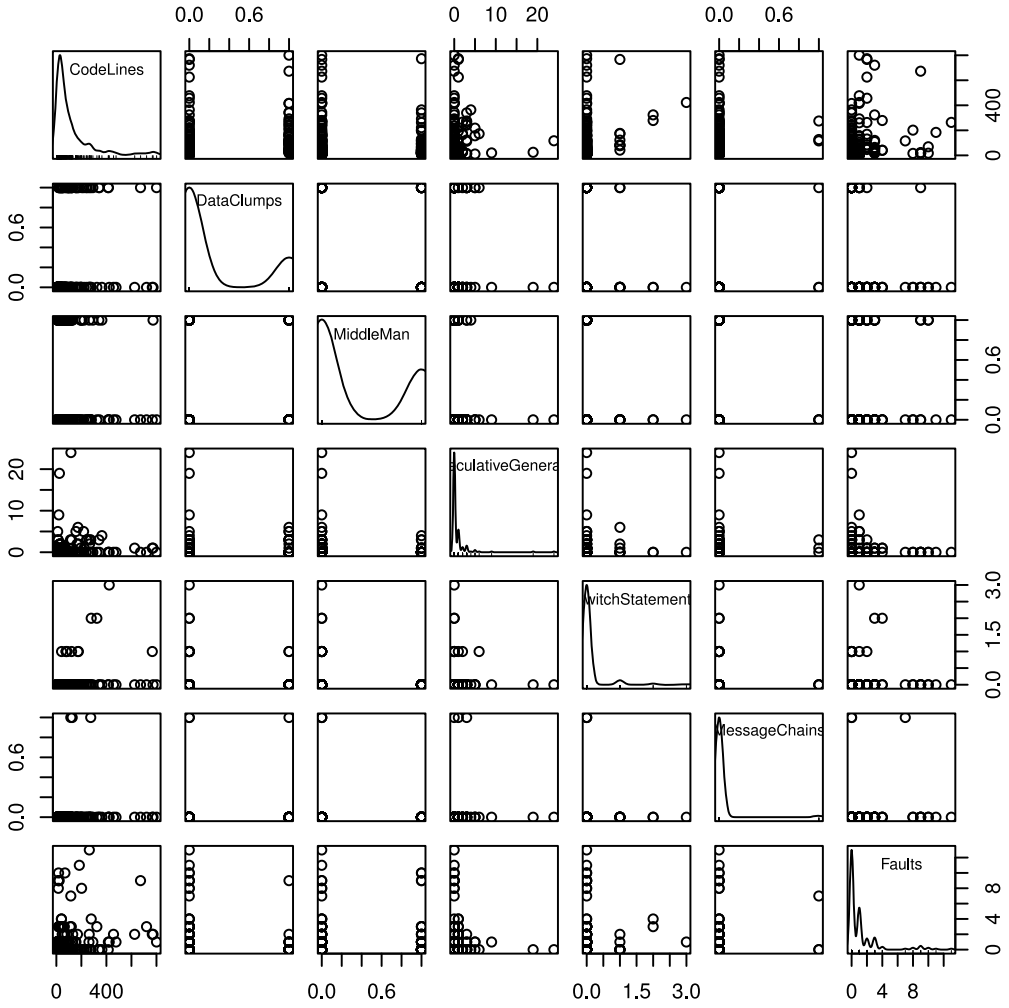


Fig. 3. A scatter plot matrix of Apache.

## D. THE CORRELATION BETWEEN INDEPENDENT VARIABLES

Table XXIX. Correlation Coefficients between the Independent Variables in Apache

| | LOC | DataClumps | MiddleMan | SpeculativeGenerality | SwitchStatements |
|---|---|---|---|---|---|
| LOC | 1.00 | 0.21 | −0.13 | 0.02 | 0.23 |
| DataClumps | 0.21 | 1.00 | −0.15 | −0.00 | −0.05 |
| MiddleMan | −0.13 | −0.15 | 1.00 | −0.16 | −0.16 |
| SpeculativeGenerality | 0.02 | −0.00 | −0.16 | 1.00 | −0.02 |
| SwitchStatements | 0.23 | −0.05 | −0.16 | −0.02 | 1.00 |

Table XXX. Correlation Coefficients between the Independent Variables in Eclipse

| | LOC | DataClumps | MiddleMan | SpeculativeGenerality | SwitchStatements | MessageChains |
|---|---|---|---|---|---|---|
| LOC | 1.00 | 0.24 | 0.27 | 0.20 | 0.37 | 0.59 |
| DataClumps | 0.24 | 1.00 | 0.08 | 0.14 | 0.07 | 0.17 |
| MiddleMan | 0.27 | 0.08 | 1.00 | 0.10 | 0.11 | 0.25 |
| SpeculativeGenerality | 0.20 | 0.14 | 0.10 | 1.00 | 0.05 | 0.22 |
| SwitchStatements | 0.37 | 0.07 | 0.11 | 0.05 | 1.00 | 0.16 |
| MessageChains | 0.59 | 0.17 | 0.25 | 0.22 | 0.16 | 1.00 |

Table XXXI. Correlation Coefficients between the Independent Variables in ArgoUML

| | LOC | DataClumps | MiddleMan | SpeculativeGenerality | SwitchStatements | MessageChains |
|---|---|---|---|---|---|---|
| LOC | 1.00 | 0.29 | −0.01 | 0.19 | 0.19 | 0.28 |
| DataClumps | 0.29 | 1.00 | 0.02 | 0.26 | 0.15 | 0.09 |
| MiddleMan | −0.01 | 0.02 | 1.00 | 0.01 | −0.03 | −0.04 |
| SpeculativeGenerality | 0.19 | 0.26 | 0.01 | 1.00 | 0.08 | 0.20 |
| SwitchStatements | 0.19 | 0.15 | −0.03 | 0.08 | 1.00 | 0.05 |
| MessageChains | 0.28 | 0.09 | −0.04 | 0.20 | 0.05 | 1.00 |

## E. DISTRIBUTION OF FAULT NUMBERS

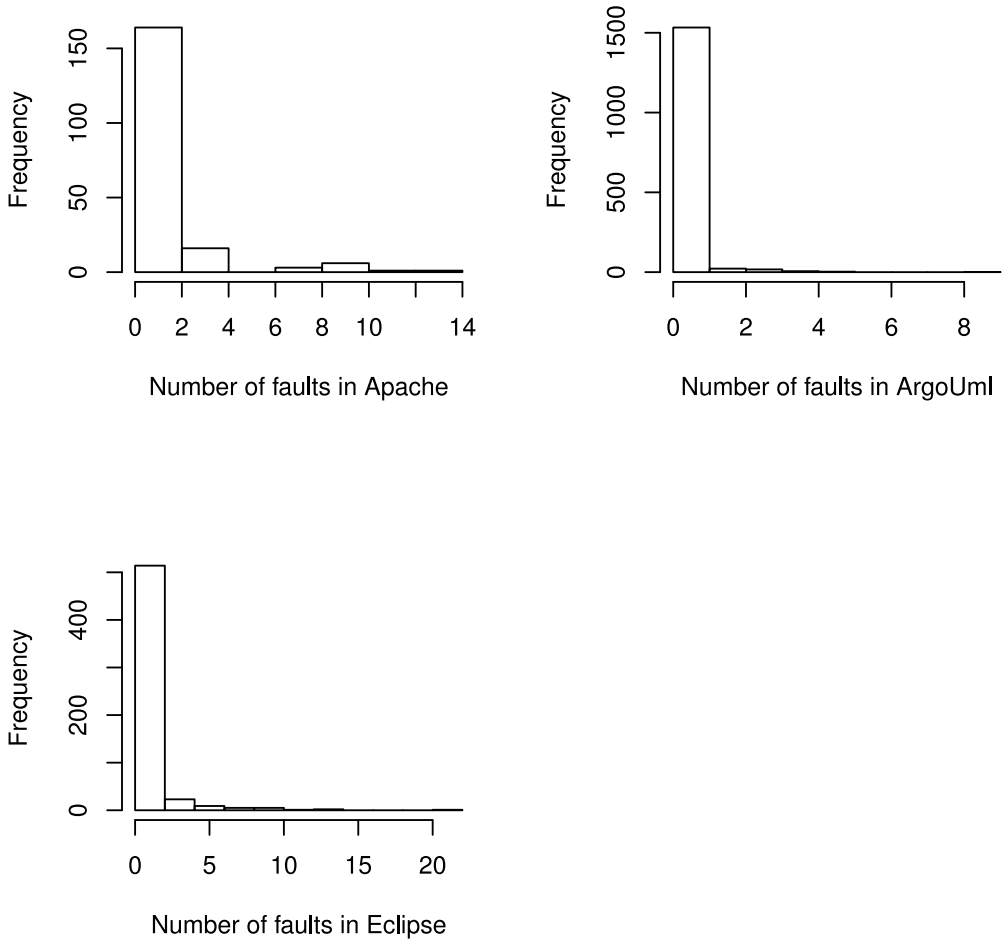


Fig. 4. Distribution of fault numbers.

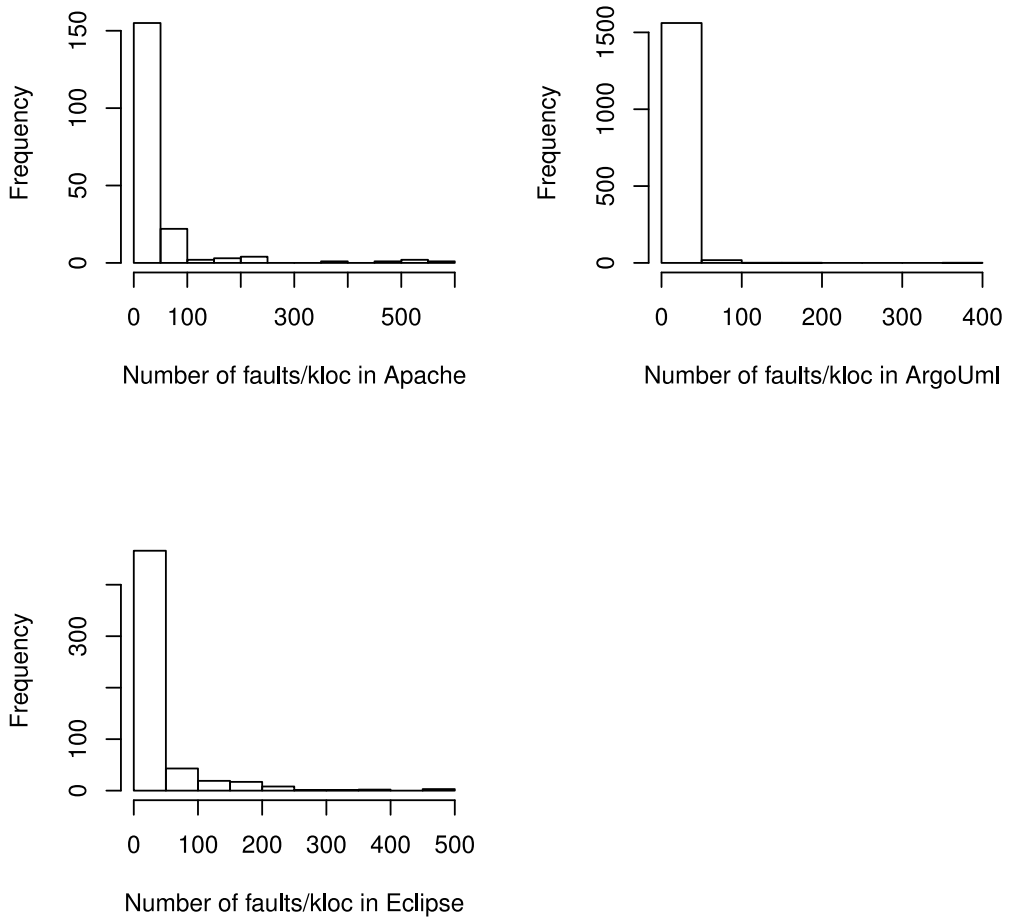


Fig. 5. Density of faults.

### F. VISUALISATION OF EACH DATASET TO THE BEST MODEL

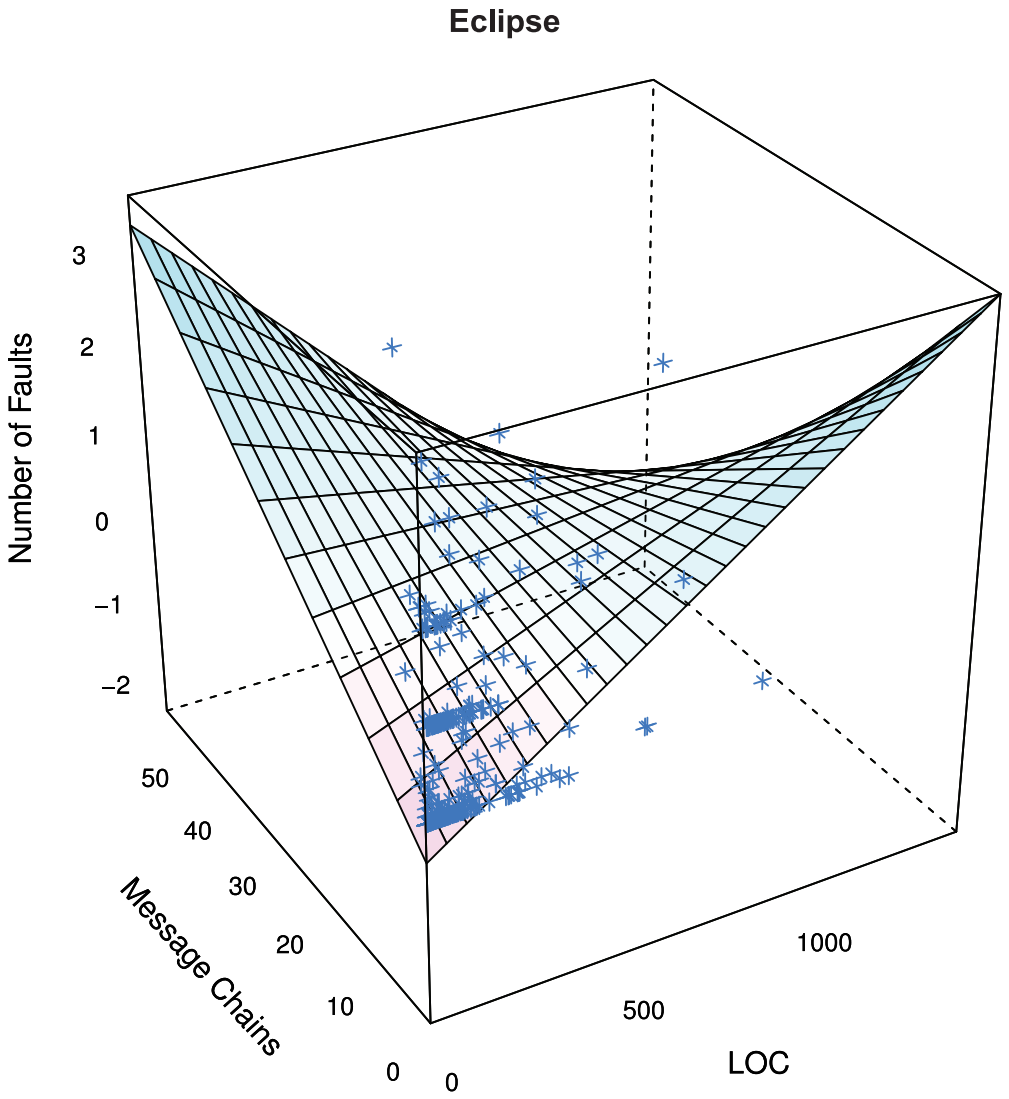


Fig. 6. Visualisation of the Eclipse data (faults, LOC, and Message Chains) with a superimposed surface which describes the predicted number of defects using the best-fit model (see Table XIX).

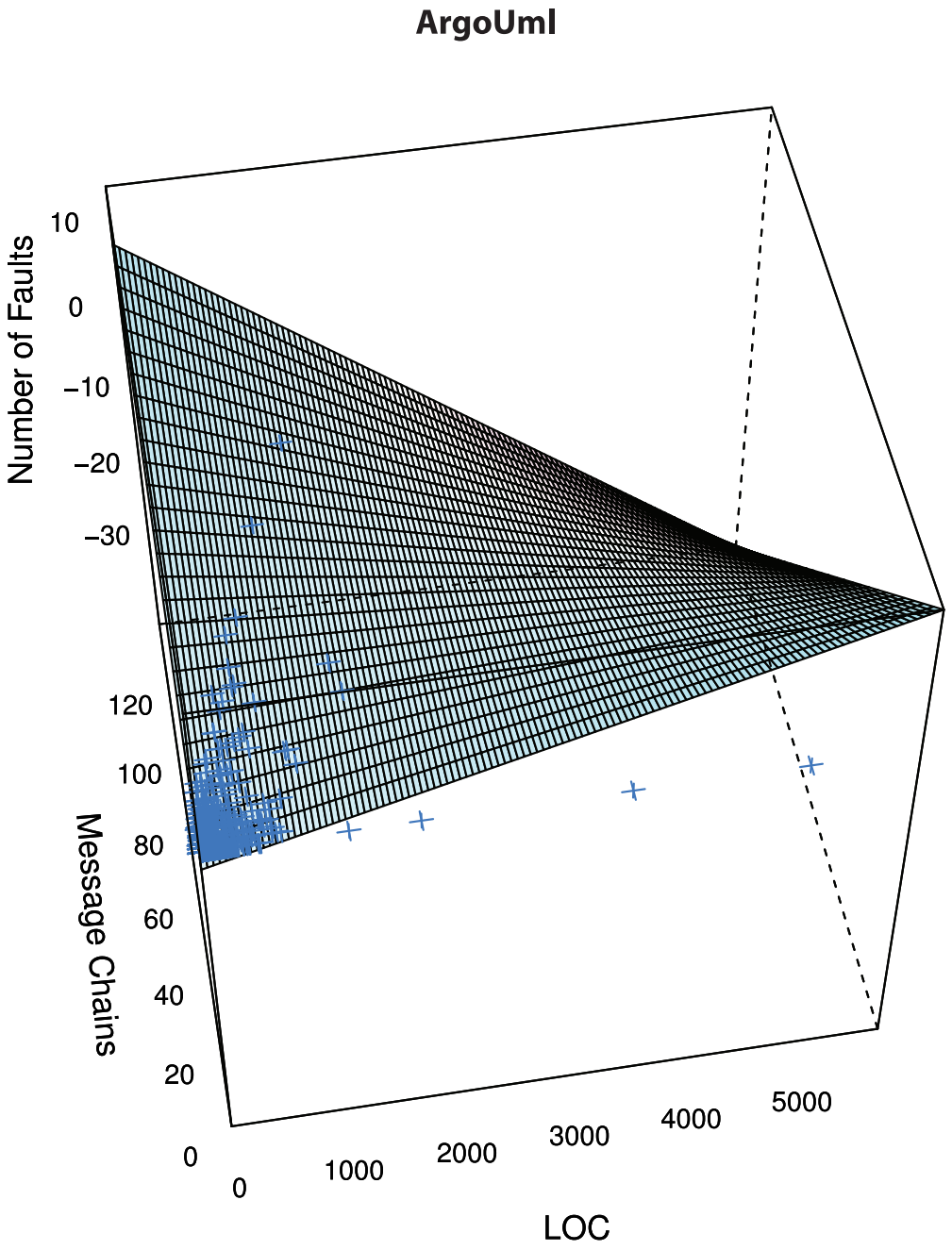


Fig. 7. Visualisation of the ArgoUML data (faults, LOC, and Message Chains) with a superimposed surface which describes the predicted number of defects using the best-fit model.

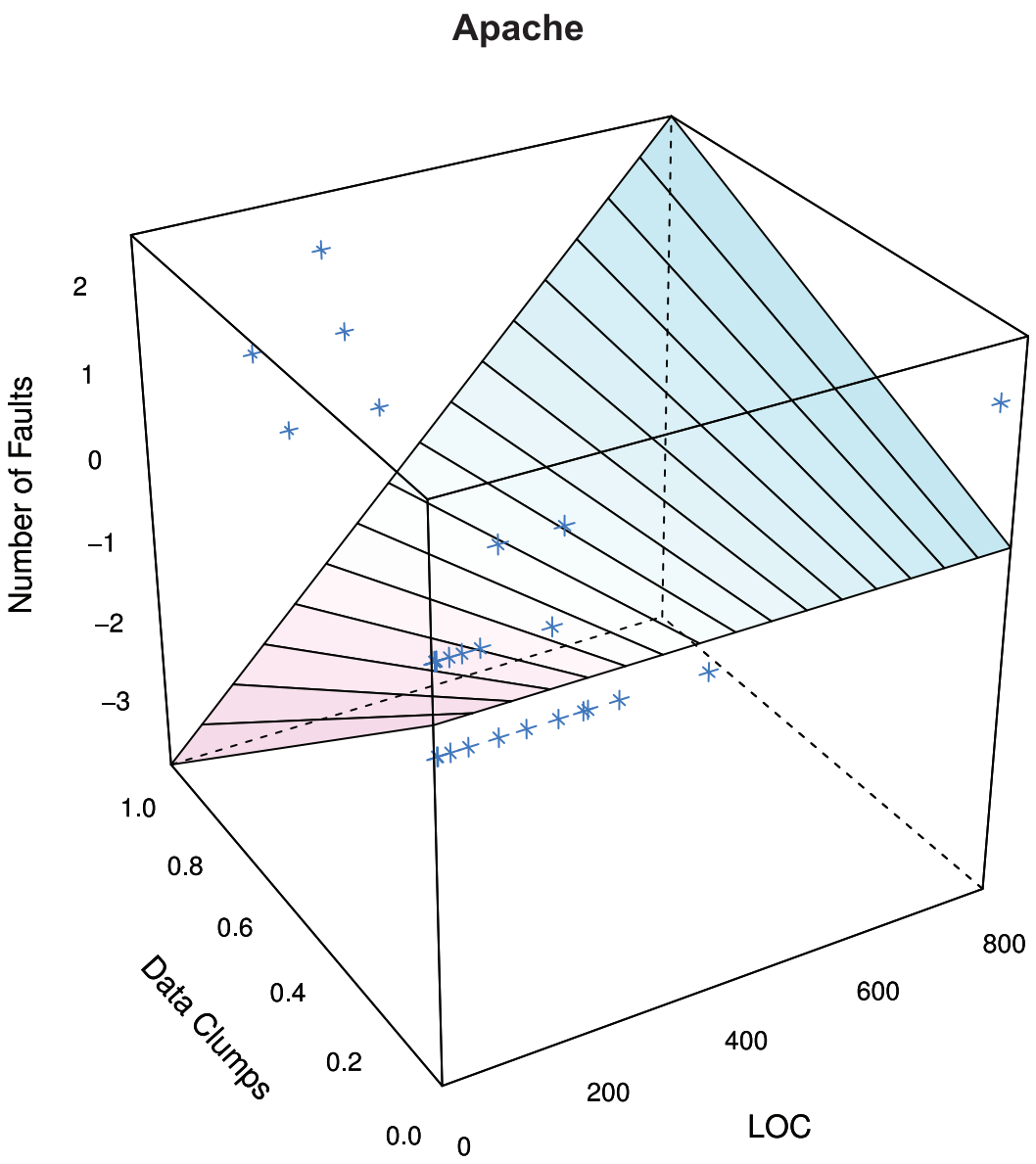


Fig. 8. Visualisation of the Apache data with a superimposed surface which describes the predicted number of defects using the best-fit model (see Table XXIII).

## ACKNOWLEDGMENTS

We thank David Randall and Thomas Shippey for the hours that they spent manually detecting smells in code as part of our tool performance evaluation. We would also like to thank Jill Stedman for her editorial help. We are also greatly indebted to the reviewers of this article for the incredibly thorough and rigorous reviews of the previous versions of this article. Without this input, the work would be much less substantial and significant.

## REFERENCES


M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*. 181–190. DOI:http://dx.doi.org/10.1109/CSMR.2011.24

R. M. Bell, T. J. Ostrand, and E. J. Weyuker. 2006. Looking for bugs in all the right places. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 61–72.

C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. In *Proceedings of the 20th International Symposium on Software Reliability Engineering*. IEEE, 109–119.

Mohamed Boussaa, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. 2013. Competitive coevolutionary code-smells detection. In *Proceedings of the 5th International Symposium on Search Based Software Engineering*. Springer, 50–65.

D. Bowes, D. Randall, and T. Hall. 2013. The inconsistent measurement of message chains. In *Proceeding of the 4th International Workshop on Emerging Trends in Software Metrics*. ACM.

William H. Brown, Raphael C. Malveau, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley.

Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Edu. Psychol. Measur.* 20, 1 (1960), 37–46.

S. Counsell, R. M. Hierons, R. Najjar, G. Loizou, and Y. Hassoun. 2006. The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. In *Proceedings of the Testing: Academic and Industrial Conference - Practice And Research Techniques (TAIC PART'06)*. 181–192. DOI:http://dx.doi.org/10.1109/TAIC-PART.2006.33

Stefany Coxe, Stephen G. West, and Leona S. Aiken. 2009. The analysis of count data: A gentle introduction to Poisson regression and its alternatives. *J. Personality Assess.* 91, 2 (2009), 121–136.

M. D'Ambros, Alberto Bacchelli, and M. Lanza. 2010. On the impact of design flaws on software defects. In *Proceedings of the 10th International Conference on Quality Software (QSIC)*. 23–31. DOI:http://dx.doi.org/10.1109/QSIC.2010.58

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2010. Improving behavioral design pattern detection through model checking. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 176–185.

Giovanni Denaro and Mauro Pezzè. 2002. An empirical evaluation of fault-proneness models. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*. ACM, New York, NY, 241–251.

Annette J. Dobson. 2010. *An Introduction to Generalized Linear Models*. CRC Press.

Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. 2006. Does God class decomposition affect comprehensibility?. In *Proceedings of the IASTED Conference on Software Engineering*. 346–355.

F. A. Fontana, E. Mariani, A. Morniroli, R. Sormani, and A. Tonello. 2011. An experience report on using code smells detection tools. In *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 450–457.

Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mantyla. 2013. Code smell detection: Towards a machine learning-based approach. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 396–399.

Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA.

Kehan Gao and T. M. Khoshgoftaar. 2007. A comprehensive empirical study of count models for software fault prediction. *IEEE Trans. Reliab.* 56, 2 (2007), 223–236.

N. Gode and R. Koschke. 2011. Frequency and risks of changes to clones. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE'11)*. 311–320.

Yann-Gaël Guéhéneuc. 2005. Ptidej: Promoting patterns with patterns. In *Proceedings of the 1st ECOOP Workshop on Building a System Using Patterns*.

Y.-G. Guéhéneuc and G. Antoniol. 2008. DeMIMA: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.* 34, 5 (2008), 667–684. DOI:http://dx.doi.org/10.1109/TSE.2008.48

T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.* 38, 6 (Nov.-Dec. 2012), 1276–1304.

T. Hall, D. Bowes, G. Liebchen, and P. Wernick. 2010. Evaluating three approaches to extracting fault data from software change repositories. In *Proceedings of the International Conference on Product Focused Software Development and Process Improvement (PROFES)*. Springer, 107–115.

Timea Illes-Seifert and Barbara Paech. 2008. Exploring the relationship of history characteristics and defect count: An empirical study. In *Proceedings of the Workshop on Defects in Large Software Systems (DEFECTS'08)*. ACM, New York, NY, 11–15. DOI:http://dx.doi.org/10.1145/1390817.1390821

Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, 485–495. DOI:http://dx.doi.org/10.1109/ICSE.2009.5070547

Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag IK Sjøberg. 2007. A systematic review of effect size in software engineering experiments. *Inf. Softw. Technol.* 49, 11 (2007), 1073–1086.

Cory J. Kapser and Michael W. Godfrey. 2008. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empir. Softw. Eng.* 13, 6 (2008), 645–692. DOI:http://dx.doi.org/10.1007/s10664-008-9076-6

Joshua Kerievsky. 2004. *Refactoring to Patterns*. Addison Wesley.

F. Khomh, M. Di Penta, and Y. Guéhéneuc. 2009a. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*. 75–84. DOI:http://dx.doi.org/10.1109/WCRE.2009.28

F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. 2009b. A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software (QSIC'09)*. 305–314. DOI:http://dx.doi.org/10.1109/QSIC.2009.47

S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 489–498.

Wei Li and Raed Shatnawi. 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J. Syst. Softw.* 80, 7 (2007), 1120–1128.

K. Lieberherr, I. Holland, and A. Riel. 1988. Object-oriented programming: An objective sense of style. *SIGPLAN Not.* 23, 11 (1988), 323–334. DOI:http://dx.doi.org/10.1145/62084.62113

Hui Liu, Xue Guo, and Weizhong Shao. 2013. Monitor-based instant software refactoring. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1112–1126. DOI:http://dx.doi.org/10.1109/TSE.2013.4

Mika V. Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empir. Softw. Eng.* 11, 3 (2006), 395–431. DOI:http://dx.doi.org/10.1007/s10664-006-9002-8

M. Mäntylä, J. Vanhanen, and C. Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*. 381–384. DOI:http://dx.doi.org/10.1109/ICSM.2003.1235447

M. V. Mäntylä, J. Vanhanen, and C. Lassenius. 2004. Bad smells - Humans as code critics. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. 399–408. DOI:http://dx.doi.org/10.1109/ICSM.2004.1357825

Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, and R. Wettel. 2005. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance-Industrial and Tool Volume*.

R. Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. 350–359. DOI:http://dx.doi.org/10.1109/ICSM.2004.1357820

Daniel McFadden. 1977. *Quantitative Methods for Analyzing Travel Behavior of Individuals: Some Recent Developments*. Institute of Transportation Studies, University of California.

T. Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (2004), 126–139. DOI:http://dx.doi.org/10.1109/TSE.2004.1265817

T. Mens, T. Tourwe, and F. Munoz. 2003. Beyond the refactoring browser: Advanced tool support for software refactoring. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*. 39–44. DOI:http://dx.doi.org/10.1109/IWPSE.2003.1231207

N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. 2010. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.* 36, 1 (2010), 20–36.

A. Monden, D. Nakae, T. Kamiya, S. Sato, and K.-i. Matsumoto. 2002. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th IEEE Symposium on Software Metrics*. 87–94. DOI:http://dx.doi.org/10.1109/METRIC.2002.1011328

M. J. Munro. 2005. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*. 15–15. DOI:http://dx.doi.org/10.1109/METRICS.2005.38

Emerson Murphy-Hill and Andrew P. Black. 2010a. An interactive ambient visualization for code smells. In *Proceedings of the 5th International Symposium on Software Visualization*. ACM, 5–14.

Emerson Murphy-Hill and Andrew P. Black. 2010b. An interactive ambient visualization for code smells. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS'10)*. ACM, New York, NY, 5–14. DOI:http://dx.doi.org/10.1145/1879211.1879216

Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 452–461.

S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. 2009. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*. 390–400.

S. M. Olbrich, D. S. Cruzes, and D.I.K. Sjøberg. 2010. Are all code smells harmful? A study of God classes and brain classes in the evolution of three open source systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. 1–10.

Rocco Oliveto, Malcom Gethers, Gabriele Bavota, Denys Poshyvanyk, and Andrea De Lucia. 2011. Identifying method friendships to remove the feature envy bad smell (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 820–823. DOI:http://dx.doi.org/10.1145/1985793.1985913

Thomas J. Ostrand, Elaine J. Weyuker, and Robert. M. Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.* 31, 4 (2005), 340–355.

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*. IEEE, 268–278.

F. Rahman, C. Bird, and P. Devanbu. 2010. Clones: What is that smell?. In *Proceedings of the 7th IEEE Working Conference Mining Software Repositories (MSR)*. 72–81. DOI:http://dx.doi.org/10.1109/MSR.2010.5463343

David Randall. 2012. A study of techniques for the definition and detection of design and code bad smells. Master's Thesis, Computer Science, University of Hertfordshire.

A. A. Rao and K. N. Reddy. 2008. Detecting bad smells in object oriented design using design change propagation probability matrix. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*. 1001–1007.

Adrian Schröter, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2006. If your bug database could talk. In *Proceedings of the 5th International Symposium on Empirical Software Engineering*, Vol. 2. 18–20.

Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. 2010. Building empirical support for automated code smell detection. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 8.

R. Shatnawi and Wei Li. 2006. An investigation of bad smells in object-oriented design. In *Proceedings of the 3rd International Conference on Information Technology: New Generations (ITNG'06)*. 161–165. DOI:http://dx.doi.org/10.1109/ITNG.2006.31

Thomas Shippey, David Bowes, Bruce Chrisianson, and Tracy Hall. 2012. A mapping study of software code cloning. In *Proceedings of the 16th International Conference on Evaluation and Assessment in Software Engineering (EASE'12)* 274–278(4).

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

Dag I. K. Sjøberg, Aiko Yamashita, Bente C. D. Anda, Audris Mockus, and Tore Dyba. 2013. Quantifying the effect of code smells on maintenance effort. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1144–1156. DOI:http://dx.doi.org/10.1109/TSE.2012.89

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proceedings of the International Workshop on Mining Software Repositories (MSR'05)*. ACM, New York, NY, 1–5.

G. Succi, W. Pedrycz, M. Stefanovic, and J. Miller. 2003. Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics. *J. Sys. Softw.* 65, 1 (2003), 1–12.

Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. 1999. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th*

*ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, NY, 47–56. DOI:http://dx.doi.org/10.1145/320384.320389

A. Trifu and U. Reupke. 2007. Towards automated restructuring of object oriented systems. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. 39–48. DOI: http://dx.doi.org/10.1109/CSMR.2007.51

S. Vaucher, F. Khomh, N. Moha, and Y. Guéhéneuc. 2009. Tracking design smells: Lessons from a study of God classes. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*. 145–154. DOI:http://dx.doi.org/10.1109/WCRE.2009.23

M. Vokac. 2004. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Softw. Eng.,* 30, 12 (2004), 904–917. DOI:http://dx.doi.org/10.1109/TSE.2004.99

Aiko Yamashita. 2012. Assessing the capability of code smells to support software maintainability assessments: Empirical inquiry and methodological approach. Ph.D. Dissertation, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo.

Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An Empirical Study. *J. Sys. Softw.* (2013).

Aiko Yamashita and Leon Moonen. 2013a. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the International Conference on Software Engineering*. IEEE Press, 682–691.

Aiko Yamashita and Leon Moonen. 2013b. To what extent can maintenance problems be predicted by code smell detection?–An empirical study. *Inf. and Softw. Technol.* 55, 12 (2013), 2223–2242.

Andreas Zeller. 2013. Can we trust software repositories? In *Perspectives on the Future of Software Engineering,* Springer, 209–215.

Min Zhang, N. Baddoo, P. Wernick, and T. Hall. 2008. Improving the precision of Fowler's definitions of bad smells. In *Proceedings of the 32nd Annual IEEE Software Engineering Workshop (SEW'08)*. 161–166. DOI:http://dx.doi.org/10.1109/SEW.2008.26

Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code bad smells: A review of current knowledge. *J. Softw. Mainten. Evol. Res. Pract.* 23, 3 (2011), 179–202.

Min Zhang, Tracy Hall, Nathan Baddoo, and Paul Wernick. 2008. Do bad smells indicate "trouble" in code? In *Proceedings of the Workshop on Defects in Large Software Systems (DEFECTS'08)*. ACM, New York, NY, USA, 43–44. DOI:http://dx.doi.org/10.1145/1390817.1390831

Yuming Zhou, H. Leung, and Baowen Xu. 2009. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. *IEEE Trans. Softw. Eng.* 35, 5 (2009), 607–623. DOI:http://dx.doi.org/10.1109/TSE.2009.32

T. Zimmermann, R. Premraj, and A. Zeller. 2007. Predicting defects for eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE'07)*. 9.

Received October 2010; revised October 2011, August 2013, January 2014; accepted February 2014