Prioritising Refactoring using Code Bad Smells

Min Zhang, Nathan Baddoo, Paul Wernick
School of Computer Science
University of Hertfordshire
Hatfield, Hertfordshire, UK
E-mail:{m.1.zhang,n.baddoo,p.d.wernick}@herts.ac.uk

Tracy Hall
Information Systems and Computing
Brunel University
Uxbridge, UK
E-mail: Tracy.Hall@brunel.ac.uk

***Abstract*—We investigated the relationship between six of Fowler et al.'s Code Bad Smells (Duplicated Code, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) and software faults. In this paper we discuss how our results can be used by software developers to prioritise refactoring. In particular we suggest that source code containing Duplicated Code is likely to be associated with more faults than source code containing the other five Code Bad Smells. As a consequence, Duplicated Code should be prioritised for refactoring. Source code containing Message Chains seems to be associated with a high number of faults in some situations. Consequently it is another Code Bad Smell which should be prioritised for refactoring. Source code containing only one of the Data Clumps, Switch Statements, Speculative Generality, or Middle Man Bad Smell is not likely to be fault-prone. As a result these Code Bad Smells could be put into a lower refactoring priority.**

Keywords- Code Bad Smells, Refactoring, Fault

I. INTRODUCTION

Software inevitably evolves after its deployment [1]. New requirements are added and existing requirements are changed. This often leads to source code becoming more complex, and results in lower software quality and reduced software maintainability [2]. To help address this problem, the refactoring technique has been introduced [3][4]. Refactoring provides methods to enhance the structure of source code without affecting the external behaviour of software. It is a key practice in eXtreme Programming [4].

Fowler et al. [4] indentify 22 Code Bad Smells as indicators of refactoring. Fowler et al. [4] suggest that Code Bad Smells can cause detrimental effects on software, and should be restructured by refactoring. For each of these 22 Code Bad Smells, Fowler et al. [4] introduce sets of refactoring methods for eliminating them.

The 22 Code Bad Smells introduced by Fowler et al. [4] cover a wide range of software problems. For example, the Switch Statements Bad Smell indicates a straightforward software coding problem where switch statements may indicate duplication in code; and the Shotgun Surgery Bad Smell represents a complicated software structure problem which means that to perform a change in software involves changing a sequence of different source code modules. Because of a lack of guidance from Fowler et al, it is difficult for software developers to make a decision on which of these 22 Code Bad Smells should be refactored from code first. As a consequence, prioritisation of Code Bad Smells for refactoring becomes important. Unfortunately there is limited previous work reporting on this. Most notably Counsell et al. [5] propose a method to reduce post-refactoring testing effort by prioritising the refactoring of Code Bad Smells. Mantyla et al. [6] present a refactoring prioritisation taxonomy to classify Fowler et al.'s Code Bad Smells according to correlations between smells.

This paper summarises our recent studies on Code Bad Smells and aims to provide guidance for software developers to prioritise Code Bad Smells for refactoring. In particular, we focus on prioritising six of Fowler et al.'s Code Bad Smells in terms of their association with software faults. These six Code Bad Smells include Duplicate Code, the number one Code Bad Smell identified by Fowler et al. [4], and the five Code Bad Smells (Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) which have never been studied individually in previous studies [7].

The reminder of this paper is structured as follows: Section II provides background on prioritising Code Bad Smells for refactoring. Section III describes the data and methods that we used for identifying Code Bad Smells and faults. Section IV presents our first investigation of prioritising five of Fowler et al.'s Code Bad Smells (Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man). Section V presents our second investigation for prioritising Duplicated Code and the other five Code Bad Smells. Section VI discusses and summarises our results. Section VII describes the limitations of our studies.

II. BACKGROUND

The 22 Code Bad Smells were informally identified by Fowler et al. in 1999 as indicators of refactoring opportunities. Fowler et al. [4] suggest that these 22 Code Bad Smells can "give you indications that there is trouble that can be solved by a refactoring". In addition to these, Fowler et al. [4] also introduce 72 refactoring methods for reducing these Code Bad Smells from code. Mens and Tourwe's [8] review suggests that Fowler et al.'s Code Bad Smells together with the refactoring methods they suggested are widely used.

However, Fowler et al. do not provide systematic guidance for software developers to prioritise the 22 Code

978-0-7695-4345-1/11 $26.00 © 2011 IEEE
DOI 10.1109/ICSTW.2011.69

Bad Smells for refactoring. As a consequence, it is difficult for software developers to decide the order in which Code Bad Smells should be refactored from code.

There are only a few attempts in the current literature of software refactoring which try to address this problem. One of the first attempts was Mantyla et al. [6]. They report a taxonomy to classify Fowler et al.'s Code Bad Smells according to correlations between Bad Smells. They propose that the 22 Code Bad Smells can be classified into seven classes: bloaters, object-orientation abusers, change preventers, dispensables, encapsulators, couplers, and others. This taxonomy provides additional information to software developers about what problems could be caused by each Code Bad Smell, and supports their decision for prioritising Code Bad Smells for refactoring. However, this taxonomy is still not proper systematic guidance for prioritising Code Bad Smells.

Other notable research in this field is from Counsell et al. [5]. Counsell et al. [5] analyse the post-refactoring test repeatability of Code Bad Smells using a taxonomy proposed by van Deursen and Moonen (VD&M) [9]. The results of their study proposed guidance for software developers on prioritising Code Bad Smells according to the difficulty associated with applying post-refactoring tests. Our approach is different from these previous approaches as we prioritise Code Bad Smells according how likely each is to be associated with software faults.

III. Code Data, Code Bad Smell Detection and Fault Identification

This section presents the code data that we used in our studies, and describes how we detected Code Bad Smells and identified faults from this code.

A. Source Code Data

We analyse two sets of source code data in this paper. The first is from the core packages of the open source project Eclipse [10]. The second is from the Apache Commons open source project [11]. Table I summarises which releases and packages we consider in our studies. Both sets of source code are written in Java.

TABLE I. A SUMMARY OF RESEARCH DATA

Project	Package	Release
Eclipse	Core	3.0, 3.0.1, 3.0.2, 3.1, 3.2
Apache Commons	Common Codec	1.3
	Common DBCP	1.2.1
	Common DbUtils	1.1
	Common IO	1.3.2
	Common Logging	1.1
	Common Net	1.4.1

Consequently, we collected 1749 source code files from the Eclipse project and 219 source code files from the Apache Commons project as our research samples.

In this paper we only focus on investigating the relationship between Code Bad Smell and faults on source code file level. In particular, we capture what Code Bad Smells exist in each of the collected source code files, and the number of faults associated with it.

B. Code Bad Smells Detection

This paper focuses on investigating six of Fowler et al.'s [4] Code Bad Smells. These six Code Bad Smells include Duplicate Code which is indicated by Fowler et al. [4] as the "number one in the stink parade", and the other five Code Bad Smells (Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) which have never been studied in details by previous studies according to the results of our recent systematic literature review on Code Bad Smells [7]. Table II presents a brief summary of the definitions of these Code Bad Smells.

TABLE II. A SUMMARY OF DEFINITIONS OF THE TARGETED CODE BAD SMELLS

Code Bad Smell Name	Fowler et al.'s Definitions
Duplicated Code	Same code structure happens in more than one place.
Data Clumps	Some data items together in lots of places: fields in a couple of classes, parameters in many method signatures.
Switch Statements	Switch statements often lead to duplication. Most times you see a switch statement you should consider polymorphism.
Speculative Generality	If a machinery was 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.
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 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.

1) *Detecting the Duplicated Code*

We used the CPD (Copy/Paste Detector) function of the PMD open source tool [12] to identify Duplicated Code from source code samples. The PMD is an open source project for searching potential problems in Java source code. The CPD function from the PMD is a Java implementation of the Rabin-Karp string matching algorithm [13] for detecting source code duplication. However, we did not consider all Duplicated Code instances in our studies. We only considered Duplicated Code which contains more than 30 source code tokens. This is because Kapser and Godfrey [14] suggest that detecting small sized Duplicated Code increases false positives in the detection result. As a consequence, we only focused on Duplicated Code which has more than 30 tokens.

2) *Detecting the Other Five Code Bad Smells*

We applied a pattern-based Code Bad Smells detection tool, developed by ourselves, to detect the other five Code Bad Smells (Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man). We developed this tool using a pattern-based Code Bad Smell detection approach. This detection approach defines Code Bad Smells as patterns in source code which is similar to Gamma et al.'s [15] definition of the GoF Design Patterns. Consequently, Code Bad Smells can be identified through searching for these patterns in source code. The more details about this tool are reported in [16][17].

C. Fault Data Collection

In this paper we applied Zimmermann et al.'s [18] approach to identify software faults from open source projects. This approach identifies software faults by locating bug fixes in the CVS repository, and then confirms these using Bugzilla bug reports. Zimmermann et al.'s [18] approach is summarised as follows:

1. Locate "bug", "fix(ed)" and "update(d)" tokens in CVS check-in comments.
2. If a CVS 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 ID.
4. The bug IDs are checked with Bugzilla bug reports. 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 the above processes for collecting fault data from Eclipse. However, because the Apache Commons project uses a different set of version control and bug tracing software from Zimmermann et al.'s [18] approach; SVN is used to perform version control and JIRA is applied to trace bugs. JIRA is a commercial product and consequently we could not access its API to capture the corresponding data. As a result, we manually collected fault data from the Apache Commons project based on Zimmermann et al.'s [18] approach.

In this paper we calculated the number of faults related to a particular source code release as the number of identified bug fixes between this selected release and the next formal release of this source code. We also considered a bug fix of a bug fix as two faults rather than a single fault. This is because several previous studies show that a bug fix is very likely to introduce new problems into software [19][20]. Hence, ignoring fixes of a fix could miss useful data.

IV. Prioritising the Five Code Bad Smells

Our first investigation prioritised the five Code Bad Smells (Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) according to their association with software faults.

A. The Association between the Five Code Bad Smells and Faults

We used the methods described in Section III to identify Code Bad Smells and faults from our collected source code samples. In particular we identified whether each of these five Code Bad Smells exists in each source code file. Because a single source code file may contain more than one smell. To eliminate the interactions between these Code Bad Smells and investigate combinations of them, we used a binary coding method to group combinations of them. This binary coding schema is shown in Table III.

TABLE III. Binary coding schema of five Code Bad Smells

Coding Profiles	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Data Clumps	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Message Chains	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Middle Man	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Speculative Generality	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Switch Statements	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Coding Profiles	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
Data Clumps	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Message Chains	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Middle Man	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Speculative Generality	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Switch Statements	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The mean numbers of faults associated to these 32 Code Bad Smell profiles captured in both the Eclipse and Apache data are presented in Figure 1. The x axis of this figure is sorted according to the number of Code Bad Smells represented by each profile. The solid line in Figure 1 indicates the mean number of faults of the zero profile using Apache data. The zero profile indicates source code not containing any of these five Code Bad Smells. The dashed line indicates the mean number of faults of the zero profile using Eclipse data.

Figure 1 reports that from both datasets the profiles 1, 2, 4, 8, and 16 which represent source code files only containing one of the five Code Bad Smells show lower mean number of faults than the zero profile. This result suggests that the source code samples which contain only one of these five Code Bad Smells is less likely to be fault-prone than the source code not containing any of these Code Bad Smells.

However, we also find that in the Eclipse dataset, profiles 18, 27, 30, and 31 have much higher mean numbers of faults than the zero profile. All of these four profiles indicate source code containing both the Message Chains and Switch Statements Bad Smells. In addition, these four profiles have included all combinations of Message Chains and Switch Statements which are possible to capture in the Eclipse dataset, this may suggest that combinations of Message Chains and Switch Statements may be fault-prone. To investigate the combination of Message Chains and Switch Statements Bad Smells further, we plot the number of faults for each investigated source code sample from Eclipse in Figure 2 and Figure 3 and separate them according to the existence of the Message Chains or Switch Statements.

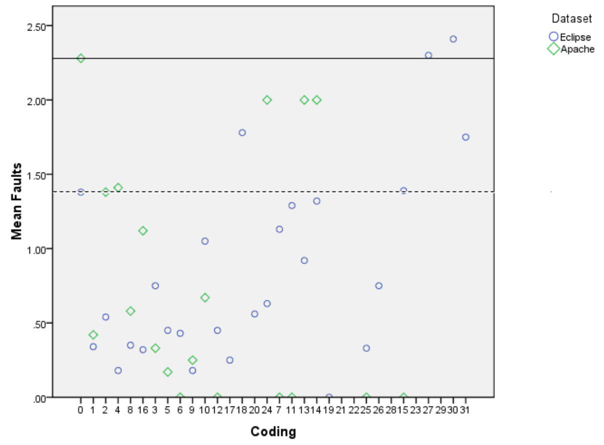

Figure 1. Mean Numbers of Faults of the 32 Code Bad Smell Profiles

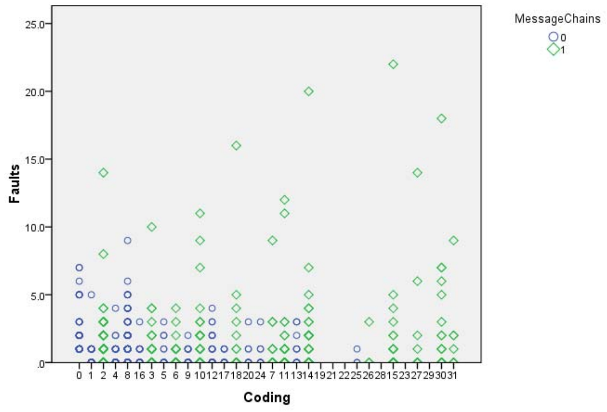

Figure 2. Numbers of faults classified by the existence of the Message Chains Bad Smell

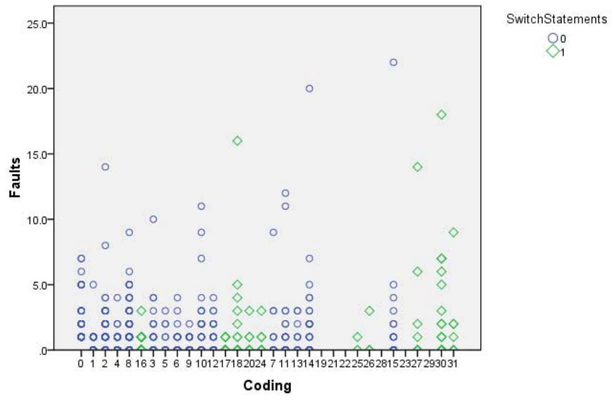

Figure 3. Numbers of faults classified by the existence of the Switch Statements Bad Smell

Figure 2 shows that all source code samples associated with more than ten faults contain Message Chains. Figure 3 indicates that the existence of the Switch Statements does not have a clear relationship with a higher number of faults. As a consequence, the high mean number of faults is likely to be associated with Message Chains rather than Switch Statements.

B. *Different Message Chains' Association with Faults*

Our first investigation suggests that all source code samples associated with more than ten faults in Eclipse contain Message Chains. This may suggest that Message Chains Bad Smell is more fault-prone than the other four investigated Code Bad Smells. However, this result does not hold for the Apache data. As a consequence, we conducted another investigation to capture what caused this difference. We manually investigated source code containing Message Chains from both the Eclipse and Apache datasets. We classified this source code according to five attributes which describe the characteristics of source code containing Message Chains. We then compared the classification results between the Eclipse and Apache samples.

1) *Sampling Strategy*

In the Eclipse dataset, 578 source code files contain Message Chains. It is not practical to manually investigate all of these. Consequently, we only selected 20 of these source code samples for detailed investigation. A systematic sampling strategy [21] was applied to select our source code samples. We selected one in every 30 cases into our investigation. In the Apache dataset, only 23 source code files contain Message Chains. We manually investigated all of them.

2) *Investigated Attributes*

This study classified source code containing Message Chains using five attributes. These attributes are summarised in Table IV. The more details about why we investigated these attributes and how we captured these attributes from source code are reported in [17].

TABLE IV. A SUMMARY OF MESSAGE CHAIN ATTRIBUTES

Attributes	Definitions
Lines of Code (LOC)	This attribute describes the size of source code. Our LOC calculation includes comment lines and blank lines.
Instances of Message Chains (IMC)	This attribute counts the instances of Message Chains in a source code file.
Max length of Message Chains (ML)	This attribute captures the maximum length of all Message Chains in a source code file.
Instances of True Message Chains (ITC)	This attribute identifies the number of true Message Chains instances in a source code file. The true Message Chains are the Message Chains created by the investigated projects instead of imported from third party libraries.
Types of True Message Chains (TTC)	We defined three possible types of Message Chains: ● **Type A Message Chains** are the Normal Message Chains, e.g. *a.getX().getY()*. ● **Type B Message Chains** are the Message Chains using temporary variables. For example: *ClassA temp=a.getX(); ClassB b=temp.getY();* ● **Type C Message Chains** are the Message Chains which at least one of the input parameters of a get method in the chain is the result of another get method, e.g. *a.getX(b.getY());*

3) Results

The results of this investigation are summarised in Table V.

TABLE V. A SUMMARY OF THE RESULTS OF THE DETAILED INVESTIGATION ON MESSAGE CHAINS

Data Set	Faults	Average LOC	Max ML	Average IMC	ITC/IMC	TTC
Eclipse	38	389	4	5	92%	A&B&C
Apache	16	481	2	2	64%	A&B&C

Note:

- **Average LOC: Total number of lines of code/number of source code file;**
- **Max ML: Max length of all Message Chains in source code;**
- **Average IMC: Total number of Message Chain instances / number of source code file;**
- **ITC/IMC: Number of true Message Chain instances / number of Message Chain instances;**
- **TTC: Contained types of true Message Chains**

Table V suggests that the average size of the investigated Apache source code is larger than the Eclipse source code. The average LOC of the selected Apache source code is 481. The average LOC of the selected Eclipse source code is 389. Table V also shows that all Message Chains in the Apache source code are short chains. The maximum length is two, which is the minimum possible length of a Message Chain. The Eclipse source code tends to contain longer Message Chains, in the 20 samples we investigated, two of them contain Message Chains with a maximum length four, and seven of them contain Message Chains with a maximum length three. Furthermore the Eclipse source code contains more instances of Message Chains than the Apache source code. The average number of Message Chain instances in the selected Eclipse source code is five, and the average number of Message Chain instances in Apache source code is only two. The Eclipse source code also contains more true Message Chains than the Apache source code. Only 64% of the Message Chains in the Apache source code are true Message Chains. In the Eclipse source code 92% of the Message Chains are true Message Chains. However both the Eclipse and Apache source code contain all three types of Message Chains.

This leads us to conclude that the different association with faults that the Message Chains Bad Smells has on Eclipse, as opposed to Apache, is not associated with the size of the investigated source code, but the characteristics of the Message Chains Bad Smells themselves. The length of the Message Chains, the number of instances of Message Chains, and whether the Message Chains are locally created or imported contribute to how Message Chains are associated with fault-proneness.

V. PRIORITISING THE DUPLICATED CODE WITH THE FIVE CODE BAD SMELLS

Duplicated Code is the first Code Bad Smells identified by Fowler et al. [4]. Fowler et al. [4] suggest that "If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them." Fowler et al. [4] also indicate that "number one in the stink parade is duplicated code". As a consequence, we investigate whether Duplicated Code is more likely to be fault-prone than the five other Code Bad Smells investigated in Section IV.

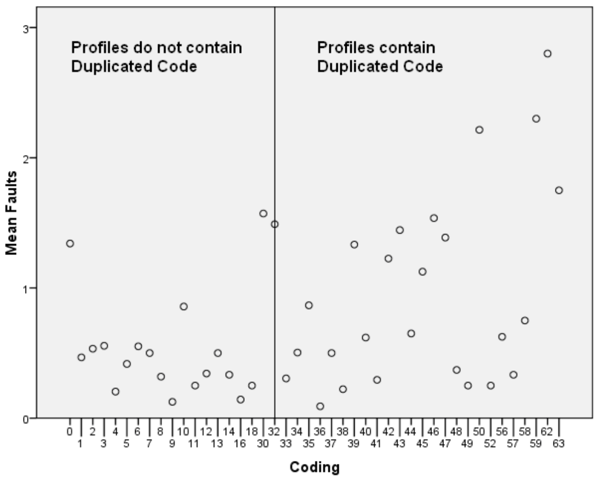

Figure 4. Mean numbers of faults of profiles considered Duplicated Code using Eclipse

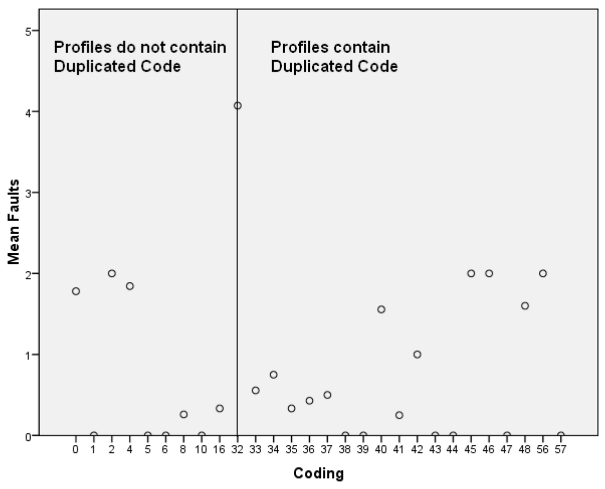

Figure 5. Mean numbers of faults of profiles considered Duplicated Code using Apache

We conducted another study to compare the Duplicated Code's association with faults with the other five Code Bad Smells. We use the CPD function of the open source tool PMD to identified Duplicated Code from both sets of our collected source code samples. Similar to previous studies, we coded the possible combination of the existence of Code Bad Smells into binary profiles. We considered six Code Bad Smells here, the five Code Bad Smells and Duplicated Code. Consequently, 64 profiles were identified. We compared the mean numbers of faults between these 64 profiles. The results of this further investigation are summarised in Figure 4 and Figure 5[1]. Profiles 0 to 31 in these figures indicate source code not containing Duplicated Code, and profiles 32 to 63 indicate source code containing Duplicated Code. In particular profile 32 indicates source code containing only the Duplicate Code.

Both Figure 4 and Figure 5 show that in general the profiles containing the Duplicated Code (profiles 32 - 63) tend to have a higher mean number of faults than the profiles which do not contain the Duplicated Code (profiles 0 - 31). In particular, the profile containing only the Duplicated Code (profile 32) has a much higher mean numbers of faults than the profile containing only one of the five Code Bad Smells (profile 1, 2, 4, 8, 16). This suggests that the source code samples containing Duplicated Code are likely to be associated with more faults than the source code samples containing the other five Code Bad Smells.

VI. Discussion and Conclusion

The results of our above studies suggest that source code containing only one of the Data Clumps, Switch Statements, Speculative Generality, or Middle Man Bad Smell is not likely to be fault prone.

Source code containing Message Chains could be associated with a very high number of faults. As a consequence, this source code should be paid more attention to. In particular, source code containing long Message Chains, a high number of instances of Message Chains, and locally created Message Chains instead of Message Chains imported from third party libraries is more likely to be associated with fault than the other source code containing Message Chains.

In addition, source code containing Duplicated Code is associated with a much higher number of faults than source code containing the other five Code Bad Smells. As a consequence, the Duplicated Code Bad Smell can be seen as the most fault-prone Code Bad Smell in our target set of six Code Bad Smells.

According these findings we summarise the following guidance for software developers to prioritise these six Code Bad Smells for refactoring.

1. Source code containing Duplicated Code should be refactored as a first priority.
2. Source code either containing more than one instance of locally created Message Chains or containing at least one locally created Message Chain whose length is more than 2 should be refactored as a second priority.
3. Source code containing the other types of Message Chains should then be considered for refactoring.
4. Source code containing only one of the Data Clumps, Switch Statements, Speculative Generality, or Middle Man Bad Smell should be refactored as a low priority.

However, we also realise that Fowler et al. identify Code Bad Smells as potential indicators of software problems. The nature of those problems is not specified. Given that the 22 Code Bad Smells are very different from one another, it is likely that they are associated with different kinds of software problems. For example, the Message Chains Bad Smell presents a highly coupled software context. This context may encourage the insertion of faults. However, the Comments Bad Smell describes badly commented source code, so is more likely to be associated with maintenance problems rather than faults. As a consequence, the guidance that we provide here applies only to refactoring intended to reduce faults. Refactoring aimed at enhancing software maintainability, is not covered by this guidance.

VII. Limitations

There are several limitations in our studies reported in this paper;

- Our guidance only considers six of Fowler et al.'s Code Bad Smells. As a consequence, we do not know what the relationship between these six Code Bad Smells and the other 16 Fowler et al.'s Code Bad Smells is in terms of their relation to software faults.
- We only consider source code from open source projects. As in all such studies our results may not be relevant to commercial software projects.
- All the source code we analysed is written in Java. Our

[1] Please notice that Figure 4 and Figure 5 do not include the mean numbers of faults of all 64 profiles. It is because some profiles can not be captured in our selected samples.

results may not be relevant to software written on other languages.

VIII. References

[1] M. M. Lehman and L. A. Belady, 1985, Program Evolution, Processes of Software Change, Academic Press.

[2] I. Sommerville, 2004, Software engineering Seventh Edition, Boston, Mass. ; London, Addison-Wesley.

[3] W. F. Opdyke, 1992, Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, PhD Thesis, University of Illinois at Urbana-Champaign.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, 1999, Refactoring: Improving the Design of Existing Code, Addison Wesley.

[5] 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," Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings.

[6] M. Mantyla, J. Vanhanen, and C. Lassenius, 2003, "A taxonomy and an initial empirical study of bad smells in code," Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on.

[7] M. Zhang, T. Hall, and N. Baddoo, 2010, "Code Bad Smells: A Review of Current Knowledge," Journal of Software Maintenance and Evolution: Research and Practice, Accepted for publication

[8] T. Mens and T. Tourwe, 2004, "A survey of software refactoring," Software Engineering, IEEE Transactions on, 30, 126-139.

[9] A. van Deursen, and L. Moonen, 2002, "The Video Store Revisited - Thoughts on Refactoring and Testing," Proc of the third International Conference on eXtreme Programming and Flexible Processes in Software Engineering XP 2002, Sardinia, Italy.

[10] Eclipse.org 2010, Eclipse, Available at: http://www.eclipse.org [Accessed 11 February, 2010]

[11] Apache.org 2010, Apache Commons, Available at: http://commons.apache.org [Accessed 11 February, 2010]

[12] Pmd.sourceforge.net 2009, PMD, Available at: http://pmd.sourceforge.net/ [Accessed 15 March, 2010]

[13] R. M. Karp and M. O. Rabin, 1987, "Efficient randomized pattern-matching algorithms," IBM Journal of Research and Development, 31, pp. 249-260.

[14] C. Kapser, and M. Godfrey, 2008, "'Cloning considered harmful' considered harmful: patterns of cloning in software," Empirical Software Engineering, 13, pp. 645-692.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, 1995, Design patterns: elements of reusable object-oriented software, Reading, Mass., Addison-Wesley.

[16] M. Zhang, T. Hall, N. Baddoo, and P. Wernick, 2008, "Improving the Precision of Fowler's Definitions of Bad Smells," 32nd Annual IEEE Software Engineering Workshop (SEW-32). Porto Sani Resort, Kassandra, Greece, pp. 161-166.

[17] M. Zhang, T. Hall, N. Baddoo, and P. Wernick, 2010, "The Impact of Bad Smells on Code," ACM Transaction of Software Engineering and Methodology. in review (available from the first author).

[18] T. Zimmermann, R. Premraj, and A. Zeller, 2007, "Predicting Defects for Eclipse," In Premraj, R. (Ed.) Predictor Models in Software Engineering, 2007, PROMISE'07: ICSE Workshops 2007. International Workshop on. pp. 9-9.

[19] J. Sliwerski, T. Zimmermann, and A. Zeller, 2005, "When Do Changes Induce Fixes?" ACM SIGSOFT Software Engineering Notes, Proceedings of the 2005 international workshop on mining software repositories MSR '05. 30. pp. 1-5.

[20] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, 2006, "Automatic Identification of Bug-Introducing Changes," Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on. pp. 81-90.

[21] M. N. K. Saunders, P. Lewis, and A. Thornhill, 2007, Research methods for business students 4th ed., Harlow, Financial Times Prentice Hall.