Improving the Precision of Fowler's Definitions of Bad Smells

Min Zhang, Nathan Baddoo, Paul Wernick
School of Computer Science
University of Hertfordshire
Hatfield, Herts, UK
{M.1.Zhang, N.Baddoo, P.D.Wernick}@herts.ac.uk

Tracy Hall
School of Information Systems, Computing & Mathematics, Brunel University
Uxbridge, Middlesex, UK
Tracy.Hall@brunel.ac.uk

***Abstract*—Current approaches to detecting Bad Smells in code are mainly based on software metrics. We suggest that these methods lack precision in detecting Bad Smells, and we propose a code pattern-based approach to detecting Bad Smells. However before such a pattern-based approach can be implemented, Fowler's original definitions of Bad Smells need to be made more precise. Currently Fowler's definitions are too informal to implement in a pattern-searching tool. In this paper we use an expert panel to evaluate our enhanced definitions for five of Fowler's Bad Smells. We use a questionnaire to survey four experts' opinions of our Bad Smell definitions. Our results show that the experts basically agree with our enhanced definitions of the Message Chains, Middle Man and Speculative Generality Bad Smells. However, there are strong disagreements on our definitions of the Data Clumps and Switch Statements Bad Smells. We present enhanced definitions on the basis of these expert opinions.**

Keywords- Coding tools and technique; programming environments/construction tools; restructuring; reverse engineering; reengineering

I. INTRODUCTION

This paper describes how we use an expert panel to evaluate our pattern-based Bad Smells definitions. Bad Smells are structures in source code informally identified by Fowler et al. [1] that can give "indications that there is trouble [in the code] that can be solved by a refactoring". They are widely used for detecting refactoring opportunities in software [2].

However, manually detecting Bad Smells is a time-consuming process and strongly depends on developers' programming experience. As a consequence, recently, several tools and methodologies have been introduced for automatically detecting Bad Smells [3-6]. Most of these methods are metric-based which identify Bad Smells using different compositions of software metrics. However, Bad Smells cannot be directly measured by software metrics. Consequently metric-based methods translate a Bad Smell into measurable code properties which are thought to be related to this Bad Smell. However, Moha et al. [7] argue that these metric-based methods are insufficient to precisely identify Bad Smells.

Hence, we propose a pattern-based approach to detect Bad Smells. The aim of our approach is to define Bad Smells as patterns of source code, so that Bad Smells can be identified through examining these patterns in source code. We report the first phase of this work here, where we have more precisely defined five of Fowler et al.'s [1] Bad Smells: Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man. Our rationale for choosing these five code smells is explained in [8]. This paper also reports our use of an expert panel to evaluate how well our definitions reflect Fowler et al.'s Bad Smells.

The rest of this paper is structured as follows: Section 2 provides an overview of our pattern-based definitions. Section 3 presents our design of an expert panel for validating our definitions. Section 4 summarises and discusses the results of our expert panel. Finally, paper is concluded and further studies are proposed in Section 5.

II. OVERVIEW OF PATTERN BASED BAD SMELL DEFINITIONS

A pattern based approach defines Bad Smells as particular patterns of source code. This is a relatively new approach to identifying Bad Smells. Previous pattern based approaches first translate source code file into a meta-model [9], and then define Bad Smells as particular structures in this meta-model. One recent study by Tourwe and Mens [10] describes how to define Bad Smells using a pattern based approach and detect them using a Logic Meta Programming language SOUL. They suggest that this is a better approach to identifying Bad Smells. However, they only demonstrate this approach for two Bad Smells and do not apply them for other Bad Smells. We have adopted a similar pattern-based approach to Tourwe and Mens [10] but have extended their approach to defining another five of Fowler et al.'s [1] Bad Smells.

This section presents an overview of our pattern-based definitions. We first describe our approach to defining pattern-based Bad Smell. Secondly, our five Bad Smell definitions are provided.

A. Definition Translating Methodology

Although Fowler et al. [1] describe Bad Smells informally, they are described in a fairly consistent way. Fowler et al. [1] first describe a generic symptom for each Bad Smell. Each symptom is then separated into several sub-situations. For each sub-situation Fowler et al. propose several refactoring methods to eliminate that Bad Smell.

In our definitions, we focus on the sub-situations described by Fowler et al. We translate each sub-situation for each Bad Smell into particular source code patterns. Each definition is translated using the following process.

1550-6215/09 $25.00 © 2009 IEEE
DOI 10.1109/SEW.2008.26

1. Read Fowler et al.'s definition of a Bad Smell.
2. If Fowler et al.'s definition separates the symptom of this Bad Smell into several sub-situations go to step 3, otherwise go to step 7.
3. Select one sub-situation to translate until all sub-situations are processed.
4. Read Fowler et al.'s descriptions of a sub-situation, if the description can be translated into source code patterns go to step 5, otherwise go to step 3 select another sub-situation.
5. If Fowler et al.'s descriptions have quantified parameters to define source code patterns use Fowler et al.'s parameters. If Fowler et al. do not provide quantified parameters define new threshold values, and refine them in implementation.

   For example, Fowler et al. defines the same 3 or 4 data items should stay together as Data Clumps. Hence, we use 3 as the threshold value to identify this Bad Smell. In contrast, Fowler et al. define Message Chains as "getThis methods that pass through many objects" [1], but they did not define a quantified parameter to describe what "many" means. Hence, we will set up a threshold value for this parameter.
6. Get to step 3.
7. Use the Fowler et al.'s generic description as a sub-situation go to step 3.

B. Pattern-Based Definitions

We have translated five Bad Smell definitions into our pattern-based definitions. These definitions are provided in Tables 1 to 5.

TABLE 1: TRANSLATION OF DATA CLUMPS DEFINITION

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. [1]
Pattern-based definition	An instance of Data Clumps Bad Smell is characterised by one of the following two situations. **Situation 1:** 1. More than three data fields stay together in more than one class. 2. These data fields should have same signatures (same names, same data types, and same access modifiers). 3. These data fields may not group together in the same order. **Situation 2:** 1. More than three input parameters stay together in more than one methods' declaration. 2. These parameters should have same signatures (same names, same data types). 3. These parameters may not group together in the same order.

TABLE 2: TRANSLATION OF MESSAGE CHAINS DEFINITION

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. [1]
Pattern-based definition	An instance of the Message Chains Bad Smell is in one of the following situations: **Situation 1:** 1. In order to access a data field in another class, a statement needs to call more than a threshold value of getter methods in a sequence. (E.g. int `a=b.getC().getD();`) 2. This method call statement and the declarations of getter methods are in different classes. **Situation 2:** 1. A statement calls more than a threshold value of temp methods in a sequence. A temp method is a method that contains at least one method call to another temp method in another class, and this method's size is not larger than a threshold value of LOC. 2. This method call statement and the declarations of temp methods are in different classes.

TABLE 3: TRANSLATION OF MIDDLE MAN DEFINITION

Fowler et al.'s definition	You look at a class's interface and find half the methods are delegating to this other class. [1]
Pattern-based definition	An instance of the Middle Man Bad Smell meets the following criteria: 1. Half of a class's methods are delegation methods. 2. A delegation method is a method that: a. Contains at least one reference to another Class. b. Contains less than a threshold value of LOC.

TABLE 4: TRANSLATION OF SPECULATIVE GENERALITY DEFINITION

Fowler et al.'s definition	If the 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. This kind of machinery includes: abstract classes that aren't doing much, methods with unused parameters, methods named with odd abstract names. [1]
Pattern-based definition	An instance of Speculative Generality Bad Smell exists if one of the following situations occurs: **Situation 1:** 1. A class is an abstract class or interface. 2. This class has not been inherited or is only inherited by one class. **Situation 2:** 1. A class contains at least one method which contains at least one parameter which is unused.

TABLE 5: TRANSLATION OF SWITCH STATEMENTS DEFINITION

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. [1]
Pattern-based definition	An instance of the Switch Statement Bad Smell meets the following criteria: 1. The code contains an instance of the *switch* key word. 2. A switch has more than two branches (including *default* statement). 3. Each branch has more than a threshold value of LOC (line of code).

III. METHODOLOGY

These pattern-based definitions are mainly based on our own interpretation of Fowler et al.'s [1] definitions of Bad Smells. To evaluate the effectiveness of these pattern-based definitions, we conducted an expert panel based study.

Expert panels are widely used in evaluating software engineering methodologies e.g. [11] as, expert opinions are an important source of information for software engineering research [12]. More particularly, Cushman and Rosenberg [13] suggest that expert opinions can provide insights that are not captured by metric-based data alone.

This section describes our methodology for conducting an expert panel.

A. Sampling Strategy

The aim of our expert panel is to capture expert opinions on our pattern-based Bad Smell definitions so that we can refine them. We need experienced Object-Oriented programming experts to take part in our panel in order to provide useful opinions. Hence, a purposive sampling strategy is adopted in this study. The purposive sampling strategy depends on the researcher's judgment to select the best fitted samples to the research [14].

Two kinds of people are involved in our expert panel: academic researchers on Object-Oriented programming and software engineers from software industry. We selected academic researchers because we think that they can theoretically examine our definitions. We selected software engineers because they can evaluate whether these definitions are likely to be useful in real world software development. In order to deeply analyse all comments from our candidates, only a small size sample is used: four experts are selected in our expert panel, two researchers and two software engineers.

B. Data Capturing Methods

We use a questionnaire to capture expert opinions. Our questionnaire is separated into two sections. The first part captures the profile of experts. In this part all questions are closed questions. The second part captures experts' opinions of our definitions. In this part, we first present each of our definitions of Bad Smells along with Fowler et al.'s [1] definition, and then several questions are asked. These questions contain both closed questions and open-ended questions. The closed questions capture whether the experts agree on our definition. If they do not agree, the closed questions also capture which parts of the definitions they do not agree with. The open-ended questions elicit experts' overall comments on our definitions. This questionnaire has been validated by a pilot study of an experienced Object-Oriented programming expert. The questionnaire has been enhanced by suggestions from this expert.

We use two strategies to deliver our questionnaires. The questionnaires to academic researchers were delivered using a structured interview [14]. We met the experts and asked them the questions in our questionnaire script face-to-face. Adopting this strategy can ensure all questions are answered by experts, and their comments are fully captured. However, because the software developers were busy with their software projects, they could not take part in our interviews. Hence, their questionnaires were delivered on-line [14]. Their questionnaires were emailed and returned electronically.

IV. RESULTS AND DISCUSSION

A. Demographic

Four experts participated in our expert panel. Their demographic profile is presented in Table 6 which shows that all experts have good knowledge of OO programming. Each of the two academic researchers has more than 20 years experience of teaching and research on OO programming. Each of the two software developers has 5 to 10 years experience of software development using OO languages. They all have a strong knowledge background to provide valuable evaluations.

TABLE 6: DEMOGRAPHIC OF EXPERTS

Expert	Roles	Year of Experiences	Knowledge of OO programming[1]
1A	Academic	21+	Excellent
2A	Academic	21+	Good
3D	Developer	5~10	Excellent
4D	Developer	5~10	Excellent

B. Overall Results

Each expert was asked for their opinions of our definitions for each of the Bad Smells in the questionnaire. The results are summarised in Table 7.

[1] Each expert was asked to select one of the following five answers to indicate their knowledge of OO programming: Excellent, Good, Reasonable, Poor, and Don't know.

TABLE 7: OVERALL RESULTS OF THE EXPERT PANEL

Bad Smell Definitions	Expert 1A	Expert 2A	Expert 3D	Expert 4D
Data Clumps	Partially Agree	Partially Agree	Partially Agree	Agree
Message Chains	Partially Agree	Partially Agree	Agree	Agree
Middle Man	Agree	Agree	Agree	Agree
Speculative Generality	Disagree	Agree	Agree	Agree
Switch Statements	Partially Agree	Disagree	Don't know	Don't know

Our results show that the software developers have stronger agreement of our definitions of Bad Smells than the academic researchers. We also found that academic researchers often refer to how students write code when describing their disagreements. Therefore, we think that this difference between researchers and developers may be caused by the different domains in which they are involved. The academic researchers normally face source code written by students. The quality of this source code is not always as good as that developed by professional developers. Consequently, academic researchers are likely to have more experience with Bad Smells than professional developers.

Moreover, our results also show that all experts agree on our definition of the Middle Man Bad Smell. We think that it may be because the Middle Man Bad Smell is defined relatively formally by Fowler et al. [1] so that our translation of this definition is straightforward. The main disagreements are in definitions of the Switch Statements, and the Data Clumps Bad Smells. Relatively, fewer disagreements exist in the Message Chains and Speculative Generality Bad Smells

C. *Reasons for Disagreement*

In our expert panel, experts were asked to provide their reasons if they disagree or partially agree on any of our definitions. Their reasons for disagreement are discussed in this section.

1) Data Clumps: There are strong disagreements in our definition of the Data Clumps Bad Smell. Three out of four experts only partially agree on our definition of this Bad Smell.

Expert 2A and Expert 3D do not agree with Situation 1 of our definition of the Data Clumps Bad Smell (See Table 1). Both suggest that not only the data fields with same signatures (same name, same data type, same access modifier), but also data fields with similar signatures (similar name, same data type, same access modifier) should be treated as Data Clumps, if they exist in more than one class. However, although we agree with this suggestion, our definitions are intended to be implemented into an automatic tool. To identify similar signatures is a subjective decision, which is hard to implement in code. Consequently, we are not going to include this suggestion in our Data Clumps Bad Smell definition.

Expert 1A argues with our definition of Situation 2 of the Data Clumps Bad Smell. Expert 1A suggests that in Situation 2 we should exclude methods inherited from parent-classes. This expert's reason is that the inheritance features of OO programming allow a method from sub-classes using the same signature to override a method from parent-classes. In this situation, we should not count the same parameters in these methods as a Data Clump, because they are not duplication. We agree with this suggestion so an additional criterion is added to our definition as in Table 1(ii).

TABLE 1(II): REFINED DATA CLUMPS DEFINTION

An instance of Data Clumps Bad Smell is characterised by one of the following two situations.

Situation 1:

1. More than three data fields stay together in more than one class.
2. These data fields should have same signatures (same names, same data types, and same access modifiers).
3. These data fields may not group together in the same order.

Situation 2:

1. More than three input parameters stay together in more than one methods' declaration.
2. These parameters should have same signature (same names, same data types).
3. These parameters may not group together in the same order.
4. These methods should not in a same inheritance hierarchy and with a same method signature.

2) Message Chains: Expert 1A and Expert 2A partially agree with our definition of the Message Chains Bad Smell. They both think our definition of "temps" in Situation 2 of this definition (See Table 2) is wrong. Expert 2A argues that defining "temps" as temp method may not indicate problems of code and it is not a good interpretation of Fowler et al.'s [1] idea, but Expert 2A cannot provide a better definition of this. Expert 1A indicates that the Message Chains Bad Smell should refer to a class A in order to access the data parts of a class B has to through sequence of other classes. This is a problem with data transmission. Hence, in defining the Message Chains Bad Smell we should consider only the source code statements related to accessing data. Because, a *getThis* method often refers to a method designed to access the data part of a class, so "temps" should refer temporary data variable which access the data part of other classes.

We think that the disagreements here are caused by Fowler et al.'s [1] ambiguous description of "temps". However, we agree with the comment from Expert 1A which better interprets Fowler et al.'s idea. Hence, our definition of the Message Chain is changed as follows.

TABLE 2(II): REFINED MESSAGE CHAINS DEFINTION

An instance of the Message Chains Bad Smell is in one of the following situations:
Situation 1:
1. In order to access a data field in another class, a statement needs to call more than a threshold value of getter methods in a sequence. (E.g. `int a=b.getC().getD();`)
2. This method call statement and the declarations of getter methods are in different classes.
Situation 2:
1. A method has more than a threshold number of temp variable.
2. A temp variable is that a variable only access data members (data fields/getter methods) of the other classes or other temp variables.

3) Speculative Generality: Expert 1A disagrees with our definition of the Speculative Generality Bad Smell (See Table 3). This expert indicates two reasons. Firstly, Expert 1A thinks that when a software project is still in the development phase, especially a project developed in parallel by two or more development teams, developers often intend to design an interface first and implement it later, so that other developers can use this interface to develop their parts of the program and do not need to wait until this interface has been fully implemented. Consequently, Situation 1 of our Speculative Generality definition is normal and should not cause problems in code. Secondly, Expert 1A argues that in Situation 2 of our definition, we should exclude the inheritance and overriding situation. Sometimes in order to reduce code duplication, similar methods in sub-classes are extracted to a parent-class. In this case, in order to provide a generalized interface, some parameters may not be used by some sub-classes.

We do not agree with the comments regarding projects still in development. We think that our definitions of Bad Smells are designed to examine the problems in stable software applications so that projects still in development should not be a driving concern. For the comment regarding inheritance, we agree that sometimes in the inheritance situation un-used parameters are reasonable, but we think that this situation should also be identified and developers should decide whether to apply refactoring. Such a situation is discussed by Fowler et al. [1] when they introduce the Remove Parameters refactoring. They indicate that in some situations putting parameters which may not be used by sub-classes in a parent-class is necessary. However, they also indicate that in many situations these parameters can be refactored using Extract Method. Consequently, we think this situation should be identified, because it can be an alarm to developers and let them consider further refactoring.

4) Switch Statements: None of the experts agree with our definition of the Switch Statement Bad Smell. Two experts, Expert 3D and Expert 4D, respond with "don't know". Their reasons are that switch statements cannot be simply treated as bad structures in code. Sometimes, the switch statements have to be used, for example, to handle keyboard inputs when using switch statements is the best solution. Expert 2A disagrees with our definition of switch statement giving a similar reason as the other two experts. This expert argues that switch statements may not cause problems in code, and suggests that only similar switch statements existing in code should indicate problems. Expert 1A partially agrees with on our definition. This expert's comment is we should also treat the *if-else* statements as switch statements, especially the *if-else* statements whose condition expression use the *instanceof* key word to check the sub-type of a data type. This expert indicates these *if-else* statements can be refactored using polymorphism.

We agree that switch statements may not cause problems in code. However, we think that Fowler et al. [1] propose the Switch Statements Bad Smell not because they think switch statements directly lead to problems in code. The real problem is that switch statements often create duplication in code which results in source code that is hard to maintain, and polymorphism is an OO programming mechanism designed to replace switch statements in code, so we should consider a polymorphism solution when we find switch statement existing in source code. We think that a switch statement is more prone to duplicated code, if it has many branches, and the code size of each branch is large. Consequently, we are not going to modify our original definition of the Switch Statements Bad Smell. However, we also think that the last expert's suggestion is reasonable; the switch statement should not only include statements starting with *switch* key words, but also the *if-else* statements. Hence, one more situation is added to our new definition of the Switch Statements Bad Smell.

TABLE 5(II): REFINED SWITCH STATEMENTS DEFINTION

An instance of the Switch Statements Bad Smell is in one of the following situations:
Situation 1:
1. The code contains an instance of the `switch` key word.
2. A switch has more than two branches (including `default` statement).
3. Each branch has more than a threshold value of LOC (line of code).
Situation 2:
1. The code contains an instance of `if-else` key word.
2. This `if-else` block has more than two branches.
3. The logic expressions in the `if-else` statements are type checking expressions using `instanceof` key words.

V. CONCLUSION AND FUTURE STUDIES

This paper presents how we use an expert panel to evaluate our pattern-based definitions of five Bad Smells. The overall results show that experts basically agree with our definitions on the Message Chains, Middle Man and Speculative Generality Bad Smells. However, there are strong disagreements on our definitions of the Data Clumps and Switch Statements Bad Smells. Our definitions are enhanced considering these expert opinions. These pattern-

based Bad Smell definitions can facilitate easier automatic detection of Bad Smells and reduce current reliance on metric-based definitions. We are now implementing these pattern-based definitions into an automatic Bad Smell detecting tool. In the future, we will apply this tool to further studies on Bad Smells. In particular, we are going to investigate how Bad Smells relate to faults in a large scale open source project.

REFERENCES

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

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

[3] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," presented at Software Maintenance and Reengineering, 2001. Fifth European Conference on, 2001.

[4] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," presented at Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, 2004.

[5] M. J. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," presented at Software Metrics, 2005. 11th IEEE International Symposium 2005.

[6] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, pp. 1120-1128, 2007.

[7] N. Moha, Y.-G. Gueheneuc, and P. Leduc, "Automatic Generation of Detection Algorithms for Design Defects," presented at Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on, 2006.

[8] M. Zhang, T. Hall, N. Baddoo, and P. Wernick, "Do Bad Smells Indicate "Trouble" in Code?," presented at International Workshop on Defects in Large Software Systems (DEFECTS 2008), Seattle, WA, USA, 2008.

[9] D. Strein, R. Lincke, J. Lundberg, and W. Lowe, "An Extensible Meta-Model for Program Analysis," *Software Engineering, IEEE Transactions on*, vol. 33, pp. 592-607, 2007.

[10] T. Tourwe and T. Mens, "Identifying refactoring opportunities using logic meta programming," presented at Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on, 2003.

[11] S. Beecham, T. Hall, C. Britton, M. Cottee, and A. Rainer, "Validating a Requirements Process Improvement Model, Technical Report 373," University of Hertfordshire. 373, 2003.

[12] S. L. Pfleeger, "What software engineering can learn from soccer," *Software, IEEE*, vol. 19, pp. 64-65, 2002.

[13] W. H. Cushman and D. J. Rosenberg, *Human factors in product design*. Amsterdam: Elsevier, 1991.

[14] M. N. K. Saunders, P. Lewis, and A. Thornhill, *Research methods for business students*, 3rd ed. Harlow: Financial Times Prentice Hall, 2003.