An Automated Code Smell and Anti-Pattern Detection Approach

Sevilay Velioğlu
IT Research & Development Center
Kuveyt Türk Participation Bank Inc.
Istanbul, Turkey
sevilay_tanis@kuveytturk.com.tr

Yunus Emre Selçuk
Computer Engineering Department
Yıldız Technical University
Istanbul, Turkey
yselcuk@yildiz.edu.tr

***Abstract*—Today, software maintenance is more expensive than development costs. As class complexity increases, it is increasingly difficult for new programmers to adapt to software projects, causing the cost of the software to go up. Therefore, it's important to produce faultless and understandable code. Moreover, software projects are not developed by one person alone; even a small-scale project needs 3 or more participants working on the code at the same time. Producing well designed code during the development stage has a significant value because this process makes software projects more understandable and leads to higher code quality. Consequently, the cost of software project maintenance will decrease. Code smells and anti-patterns are symptoms of poorly designed code. The aforementioned tendencies of software projects increase the possibility of poor implementations and code imperfections. Therefore it is necessary to detect and refactor poorly designed code. This paper describes an attempt to achieve their detection.**

Keywords—code smells, anti-patterns, automatic detection, brain method, data class

I. INTRODUCTION

Code smells are defined as the recurring signs of weak design and coding by Fowler [1]; whereas, anti-patterns commonly provide incorrect and risky solutions to existing problems [2]. There are lots of investigations and solutions concerned with code smell detection issues. Discovering code smells manually is difficult for developers. Furthermore, manual detection increases the cost of development. As system size increases, the cost of locating code smells in the source code also increases. As a result, using an automated tool for software maintenance will counter these size disadvantages [3].

It is highly probable that many developers will participate in inherited software systems instead of working on new projects; therefore, it is crucial to create understandable projects in the first place. Fixing software bugs and refactoring poor design during the development stage will make applications more understandable. Consequently, developers will produce more work in a shorter time, and this approach will reduce the cost of software maintenance.

The aim of this study is to develop a tool to locate code smells and anti-patterns that may lead to deeper problems in the future. This paper addresses two distinct stages: first defining the factors that lead to code smells and anti-patterns; then exploring them by using these factors.

The motivation for this paper is to provide a tool (called Y-CSD), that reduces code smell and anti-patterns to increase the code quality; thus, decreasing software maintenance costs and helping new developers adapt to existing projects.

II. RELATED WORK

A. Proposed Methods

Code smells are weak solutions; consequently, they are symptoms of poor coding. In the event that any code requires a redesign of its structure, we can say that there are some code smells located in it [4]. Many authors have contributed various code smell and anti-pattern definitions to published literature. For example, Fowler states that code smells are indicators of deeper problems in software systems [1].

Refactoring is a modification technique to make source code more comprehensible and minimize coding faults without changing system behavior. It is easier to introduce refactoring in a step-by-step fashion to source code. Some refactoring examples are to push up variables from one class to another, extracting new methods from long methods, pulling part of the source code up from a child class to its parent class, or conversely, push part of the source code down from the parent class to its child class. Refactoring, executed in the maintenance phase, or in other development phases of the software, increases the quality of the source code and prevents defects that may cause future problems [1].

The anti-pattern phenomenon was discovered by Koenig [2]. An anti-pattern is supposed to be a design pattern, but it is not. Unlike design patterns, anti-patterns express some recurring faults in software development. Knowledge of anti-patterns eases the discovery and prevention of design problems [5].

B. Tool Contributions

There are various tools that can find bad smells and/or anti-patterns. Some tools also provide refactoring suggestions. The tools that we were able to find are PMD [6], Essere CSD[7] and CBS Detector [8].

PMD is source code analyzer that can find defects by using rules, and it generates abstract syntax trees during the detection process. It can be integrated with many IDE (Integrated Development Environment) tools including Eclipse and NetBeans. It has the ability to find out the following:

- Empty try/catch/finally/switch blocks
- Unused local variables and parameters
- Redundant string usage
- Unnecessary if statements
- Using *for loop* instead of *while loop*.

The Essere Code Smell Detector finds java code smells by using machine-learning techniques. It has been integrated with only Eclipse IDE so far, and it focuses on the following code smells:

- Data Class: Such classes do not have any method.
- Lazy Class: Classes that contain only limited methods are called lazy classes. If the ratio of the total number of member fields and methods in a class in comparison to the total number of members in a package is smaller than 2/3, then such classes are called lazy classes.
- Middleman: The middleman takes over the intermediary role between two or more classes. It provides indirect communication between objects. If the number of delegate methods of a class is similar to its non-delegate methods, then this class is called middleman.
- Long Parameter List: If any method of a class has more than a few parameters, then it is called a Long Parameter List. This kind of code smell makes it difficult for programmers to understand such methods.
- Feature Envy: If a method accesses other classes' members instead of its own members, then this behavior is called Feature Envy.

The CBS Detector is currently not integrated with any IDE tool and has the ability to detect the following code smells:

- Middleman: Refer to previous description above.
- Data Clumps: These are the same few data items that are constantly passed around together as method parameters. They should have been modeled as class member fields in the first place.
- Switch Statements: Excessive use of switch statements often indicates that the use of polymorphism has been overlooked.
- Speculative Generality: Also known as premature generalization, this smell indicates code generated to support anticipated future features that has never been realized.
- Message Chains: A message chain occurs when a client requests another object, which requests yet another one, and so on. This multiple level of requests can indicate unnecessary coupling. Any changes to the class structure increase the threat that such code will break, increasing the maintenance cost.

C. Summary

All of these approaches use different techniques to contribute automated code smell detection. However, none of them provides a feature of constitution dynamic thresholds.

III. PROPOSED METHOD

This paper aims to find Brain Method and Data Class code smells by using software metrics. Using software metrics provides a way to achieve more accurate results.

A new tool was developed to demonstrate detected code smells to end users. In addition, this tool provides a feature of constructing thresholds by users.

A. Detected Anti-Patterns

1) Brain Method:

A method that has grown excessively over a period of time by adding new functionality is called a brain method. This leads to greater complexity and increased maintenance [9].

2) Data Class:

A class that provides data instead of functionality is called a Data Class. Their encapsulation level is low, and their functionality is insufficient [1]. This code smell hampers organization of code and unnecessarily increases coupling because the required functionality will eventually be coded elsewhere.

B. Detection Method

Code smells in a source code can be detected manually or automatically. Manual detection is more accurate if people with expert knowledge are assigned to it but represents inefficient use of resources in terms of people and time. By using automatic detection tools, efficiency can be increased. Considering the ever expanding size of software projects and the importance of efficiency, an automated model is proposed here.

We preferred to use structural analysis in our work. The proposed model requires both training and testing project work to be performed. Firstly, training projects are used to establish thresholds for detecting code smells. The thresholds obtained are kept in a flat file for reuse at a later stage.

The proposed tool saves the following measurements in a text file to use them for detection: Total lines of code, average lines of code, total number of methods, average number of methods, total number of if/else conditions, average number of if/else conditions, total number of while conditions, number of variables, number of accessors, number of try/catch blocks and number of and/or operators.

We preferred a dynamic run-time calculation of upper and lower thresholds from a pool of training projects in order to automate code smell detection. The following metrics were used to detect code smells:

- CYCLCO (McCabe's Cyclomatic Number): This calculates the number of linear independent operations. This metric is used to find Brain Method code smells.
- LOC (Lines of Code): This calculates the number of lines in a class. Empty lines are excluded. This metric is used to find Brain Method code smells.
- NOAM (Number of Accessor Methods): This calculates the number of accessor (*get/set*) methods. This metric is used to find Data Class code smells.
- NOM (Number of Methods): This calculates the number of methods in a class. This metric is used to find Brain Method code smells.
- NOPA (Number of Public Attributes): This calculates the number of public variables in a class. This metric is used to find Data Class code smells.
- WMC (Weighted Method Count): This calculates the complexity of a method. This metric is used to find Data Class code smells.
- MAXNESTING (Maximum Nesting Level): This calculates the depth of condition statements. This metric is used to find Brain Method code smells.
- NOAV (Number of Accessed Variables): This calculates the number of variables in a method. This metric is used to find Brain Method code smells.
- CBO (Coupling Between Object Classes): This calculates coupling between classes. This metric is used to find Data Class code smells.

Our training set contains 8 projects that are classified as well designed by the internal IT department of a private finance company; and as a whole, the set has the following properties:

- 32645 lines of code
- 1397 methods
- 2386 variables
- 2416 *if/else* conditions
- 79 *while* loops
- 28 *for* loops
- 60 *switch/case* statements
- 107 *try/catch* statements
- 733 *and/or* operations
- 2983 accessor methods
- 16 maximum nesting levels
- CBO is less than 4.

1) Brain Method Detection Strategy:

Our Brain Method detection strategy uses thresholds determined by the training set. Brain methods are extremely large methods. To detect these instances, we find the size of each method by using the LOC metric. A brain method contains more than one task, so the probability of being a brain method is proportional to size. Results of this metric are compared with the established upper and lower limits; if the current method value exceeds the upper limit, the detection score is incremented by one.

Another symptom associated with the Brain Method is a tendency to include a large number of local variables. The more the number of local variables exceeds the upper limit, the closer that method tends to become an instance of a Brain Method code smell. To calculate the number of local variables we use the NOAV metric. If the result of the NOAV metric for a method exceeds the upper limit, then the detection score is incremented by one.

Thirdly, we analyze methods that contain *if/else* conditions. The larger the number of conditions that exceed the upper limit, the closer that method tends to become an instance of a Brain Method code smell. CYCLCO is used to measure condition complexity. If the result of the CYCLCO metric for a method exceeds the upper limit, then the detection score is incremented by one.

Lastly the degree of nesting level is analyzed. The nesting level is directly proportional to a Brain Method instance. The MAXNESTING metric is used for computing the depth of nested conditions. If the result of the MAXNESTING metric for a method exceeds the upper limit, then the detection score is incremented by one.

If the total detection score is equal to four then the method examined is flagged up as a Brain Method, and it is added to the Brain Method pool.

2) Data Class Detection Strategy:

Our Data Class detection strategy also uses thresholds determined by the training set. Data Classes provide data instead of functionality, therefore we focus on the property of classes by using specific metrics such as NOAM, NOAV and WMC. The number of variables and *get/set* methods give information about the tendency to be a data class, so NOAM and NOAV metrics are used to detect this code smell. The NOAV metric calculates the number of variables, and the NOAM metric calculates the number of *get/set* methods in a class. If the result of the NOAV+NOAM metric for a class exceeds the lower limit, and if the result of the WMC metric for a class is smaller than the lower limit, the detection score is incremented by 1, and it is then multiplied by 0.60. If the previous condition is false, then we calculate a second condition: If the result of the NOAV+NOAM metric for a class exceeds the upper limit, and if the result of the WMC metric for a class is smaller than the upper limit, then the detection score is incremented by 1 and multiplied by 0.60.

Another hint providing information about the tendency to be a data class is the coupling between classes. The CBO metric is used for analyzing how often classes are coupled. The CBO for a class is calculated and then multiplied by 0.40.

The sum of these results is used to analyze the class according to the following rule: if the total score is equal to 5 then the inspected class is flagged up as a Data Class, and it is added to the Data Class pool.

C. *Implementation*

This section gives details about our tool implementation, called Y-CSD. Y-CSD is a windows form application coded in C#. Through supplied user interfaces, end users can easily analyze code smell results.

In our implementation, the NRefactory library is used to analyze system source codes and to obtain both syntax and semantics of C# projects. NRefactory generates an abstract syntax tree for given code blocks. This tree is filtered to obtain specific leaves that are named as descendants.

IV. EVALUATIONS

IPlasma [9] and Essere [7] tools are used for comparisons with Y-CSD tool. A C# to Java converter is used to convert C# test projects to java source code to allow the evaluation of various applications that support different platforms. The Visual Studio Code Metric Calculator is used to find high-quality projects by ordering them according to their CBO value.

While iPlasma and Essere tools check Java source code, Y-CSD checks C# source code. Therefore we have used a C# to Java converter tool to compare these three tools with each other. The source code of the test projects in C# are converted to Java source code by making use of that converter tool, so that roughly the same source code is analyzed for comparison.

In order to compare the accuracy, the expert opinion of three software developers has been used. We chose these developers for their knowledge and experience to get the most accurate responses possible. The first developer is an expert in C# development. He has been working for six years in the Kuveyt Turk Participating Bank as a software developer. The second developer is working as a software solution architect in EAE Technology. He establishes limits and standards for other developers so that projects have a similar appearance to each other. The third developer is knowledgeable about Java and PHP languages, and she is an expert at coding in C#. She has been working for 5 years as a C# developer in the Kuveyt Turk Participation Bank.

A. *Comparing Results*

Table I shows the expert opinion and tool evaluation of some classes of source code for Data Class code smells. Results that contradict with expert opinion are given in italics. According to the assessment of the senior developers, only 1 of 18 classes are considered to have a Data Class code smell.

TABLE I. TEST RESULTS OF CLASSES FOR DATA CLASS CODE SMELL

Class Name	**Considered as Data Class (expert opinion)**	**Y-CSD**	**iPlasma**	**Essere**
CustomerDefinitionRequest	Y	Y	*N*	*N*
KTVPosAddress	Y	Y	Y	*N*
KTVPosMessage	Y	Y	*N*	Y
KTVPosProduct	Y	Y	Y	Y
OrderRequest	Y	Y	Y	*N*
CreditCardRequest	N	N	*Y*	*Y*
CustomerDefinitionContract	Y	*N*	*N*	Y
OrderContract	Y	*N*	*N*	Y
VPosIndependentDrawbackReversalTransactionRequest	Y	Y	Y	*N*
VPosIndependentDrawbackTransactionRequest	Y	Y	Y	Y
VPosManuelClosePreAuthTransactionRequest	Y	Y	Y	*N*
VPosManuelSettlementTransactionRequest	Y	Y	Y	*N*
VPosManuelSaleReversalTransactionRequest	Y	Y	Y	*N*
VPosNonThreeDSaleTransactionRequest	Y	Y	Y	*N*
VPosManuelDrawBackTransactionRequest	Y	Y	Y	*N*
VPosManuelPreAuthReversalTransactionRequest	Y	Y	Y	*N*
VPosManuelPreAuthTransactionRequest	Y	Y	Y	*N*
MerchantDetailsContract	Y	*N*	*N*	Y

Table II shows accuracy results of different tools for Data Class detection.

TABLE II. DATA CLASS DETECTION ACCURACY RESULTS

	Y-CSD		***iPlasma***		***Essere***	
	Positive	***Negative***	***Positive***	***Negative***	***Positive***	***Negative***
True	14	1	12	0	6	0
False	0	3	1	5	1	11

Table III shows the expert opinion and tool evaluation of some classes of source code for the Brain Method anti-pattern. Results that contradict expert opinion are given in italics. According to the assessment of the senior developers, 7 methods are considered to have that smell.

TABLE III. TEST RESULTS OF CLASSES FOR BRAIN METHOD ANTI-PATTERN

Class Name	**Considered as Brain Method (expert opinion)**	**Y-CSD**	**iPlasma**
GetMerchantOrderDetail	Y	Y	*N*
OrderByMerchantOrderId	Y	Y	*N*
SelectByKey	Y	Y	*N*
NonThreeDPaymentByMaskedCards	N	*Y*	N
SaleReversalByMaskesCard	N	*Y*	N
DrawBackByMaskedCard	N	*Y*	N
PartialDrawBackByMaskedCard	N	*Y*	N
GetCreditCard	Y	Y	*N*
NonThreeDPayment	N	N	N
PartialDrawback	N	N	N
deDrawBack	N	N	N

Table IV shows accuracy results of different tools for Brain Method detection:

TABLE IV. BRAIN METHOD DETECTION ACCURACY RESULTS

	Y-CSD		***iPlasma***	
	Positive	***Negative***	***Positive***	***Negative***
True	4	3	1	7
False	4	0	0	3

Table V further summarizes all the results so far, based on accuracy. As the Brain Method detection is not supported by the Essere extension, that test is applied to the iPlasma and Y-CSD tools.

TABLE V. ACCURACY COMPARISON

Code Smell Name	**Y-CSD**	**iPlasma**	**Essere**
Data Class	83.3% (15/18)	66.7% (12/18)	33.3% (6/18)
Brain Method	63.6% (7/11)	72.7% (8/11)	Not Available

V. FUTURE WORK

Our proposed methods for Data Class and Brain Method detection show acceptable accuracy, although there is room for improvement in terms of accuracy, new features and new code smells/anti-patterns. Notable future improvements to our tool, Y-CSD, can be listed as follows:

- Ability to work on multiple object oriented languages.
- Ability to work as an extension to Visual Studio.
- Ability to suggest refactoring steps to remove detected code smells.
- Ability to draw graphs that show developers' contributions on internal software quality.

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[2] A. Koenig, "Patterns and antipatterns," Journal of Object-Oriented Programming, 8(1), pp. 46-48, 1995.

[3] M. O'Keeffe and M. O´Cinneide, "Search-based refactoring for software maintenance," Journal of Systems and Software, 81(4), pp. 502-516, April 2008.

[4] P.A. Laplante, What Every Engineer Should Know About Software Engineering, CRC Press, 2007.

[5] L. Rising, The Patterns Handbook: Techniques, Strategies and Applications, Cambridge Univ. Press, 1998.

[6] PMD, https://pmd.github.io/, last visited: Nov. 28th, 2016.

[7] Essere Code Smell Detector, http://essere.disco.unimib.it/reverse/CodeSmellDetector.html, last visited: Nov. 28th, 2016.

[8] Code Bad Smell Detector, http://sourceforge.net/projects/cbsdetector, last visited: Nov. 28th, 2016.

[9] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2006.