Feature Envy Factor

A Metric for Automatic Feature Envy Detection

Kwankamol Nongpong
Department of Computer Science
Assumption University
Bangkok, Thailand
Email: kwan@scitech.au.edu

***Abstract*—As a software system evolves, its design get deteriorated and the system becomes difficult to maintain. In order to improve such an internal quality, the system must be restructured without affecting its external behavior. The process involves detecting the design flaws (or code smells) and applying appropriate refactorings that could help remove such flaws. One of the design flaws in many object-oriented systems is placing members in the wrong class. This code smell is called Feature Envy and it is a sign of inappropriate coupling and cohesion. This work proposes a metric to detect Feature Envy code smell that can be removed by relocating the method. Our evaluation shows promising results as the overall system's complexity is reduced after suggested Move Method refactorings are applied.**

Keywords-feature envy; code smells; refactoring; design flaws; software quality; software metric

I. INTRODUCTION

The evolution of a software system is a long and continuous process. Software system usually goes through a series of small and big changes over a period of time. When a software system gets larger, new features are added to the system, its design generally gets worsened. It gets more complicated and difficult to understand. Furthermore, if the design is not regularly revised, the system will become difficult to manage and maintain.

To promote continuous design revision, the process of locating and removing code smells should be transparent and seamless to the software developer. The process includes identifying code smells and fixing them using behavior-preservation transformations or refactoring. Refactoring is usually initiated by the developer. Most software developers only refactor their code when it is really necessary because this process requires in-depth knowledge of that particular module of the software system. While many experienced developers can recognize the pattern and know when to refactor, novice programmers may find this process very challenging.

Fowler et al. [4] describe a set of code smells and how to resolve them with corresponding refactorings. A number of studies [16, 19] also confirm that code smells have a great impact on software maintainability and proper use of refactorings can actually help improve the software qualities of the system.

II. FEATURE ENVY

One of the main design flaws in object-oriented systems is misplacing the class members or making the class responsible for things that should be handled by other classes. In object-oriented systems, classes must be loosely coupled and highly cohesive. The code smell that involves wrong placement of the class members is called Feature Envy.

Feature Envy code smell is a sign of inappropriate coupling between classes. It occurs when a class member is more interested in some other classes than the class that it is currently defined in. The higher the coupling between classes, the higher the number of classes are affected when changes are made to the system. A small premeditated change in a highly coupled system could result in a long series of unanticipated changes in a lot of classes. Hence, class interdependence should be kept to the minimum if possible.

The rule of thumb for this code smell is that if the feature does not really belong to the class it is defined in, it should then be given a new home.

A. Coupling and Cohesion Measures

Since Feature Envy code smell concerns coupling and cohesion, we look at some existing coupling and cohesion measures. Chidamber and Kemerer [2] defined coupling as a situation when methods declared in one class use methods or instance variables defined by the other classes. They propose a metric called Coupling Between Objects (CBO) counts the number of other classes to which it is coupled.

There are several classes of cohesion measures: structural metrics [2, 5], slice-based metrics [10, 13], and information retrieval approach [8, 14]. Lack of Cohesion in Methods (LCOM) and its variants are the most investigated structural cohesion metrics.

It has been observed that the granularities of existing structural coupling and cohesion metrics are at the package or the class level but Feature Envy occurs inside the class, or more specifically, at the method level. Hence, such metrics are not applicable for use in Feature Envy detection.

978-1-4799-6049-1/15/$31.00 ©2015 IEEE

B. Identifying Feature Envy Candidates

There are many attempts on detecting Feature Envy code smell candidates and recommending refactorings. One of the approaches that are used to identify Feature Envy is based on calculating two values which are 1) similarity between methods and 2) the distance between the method and the target class candidates [11, 15, 17].

Tsantalis and Chatzigeorgiou identify Move Method refactoring opportunities by computing similarity between a method and a class using entity sets [17]. The entity set of a method contains entities that it accesses while the entity set of a class contains all members that belong to the class (excluding getters and setters). The inner and outer entity distances are calculated. While inner entity distances should be small to achieve high cohesion, outer entity distances should be large for low coupling. Their entity placement metric of a class is the ratio of average inner to average outer entity distances. If the ratio is high, the class may not be cohesive or it is highly coupled with other class.

The approach proposed by Sales, Terra, Miranda and Valente involves evaluating dependency sets for a given method *m* in a class *C* [15]. Two average similarity coefficients are computed. The first average similarity identifies the dependencies between m and the remaining methods in the class. The second average similarity determines the dependencies between m and methods in another class C_i. If the first average is lower than the second average, then *m* is more similar to methods in other class C_i than its current class *C* and C_i could be a new home for *m*.

Oliveto, Gethers, Bavota, Poshyvanyk and De Lucia suggest a different technique to identify Feature Envy bad smell by identifying method friendships [11]. According to their viewpoint, methods and classes are very similar to groups of people. In this approach, the degree of similarity among methods in the system is determined and friendships among these methods are ranked.

One other challenge of code smell detection and removal is the fact that more than one refactorings may be recommended. Tsantalis and Chatzigeorgiou introduce an approach to rank suggested refactorings by performing code smell evolution analysis [18]. According to historical volatility model, code fragments that have been undergone series of changes in the past are highly likely to be changed in the future.

III. PROPOSED FEATURE ENVY METRIC

This work proposes a quantitative measure of feature envy that considers both internal and external class dependencies.

A. The Metric

In metric suite by Chidamber and Kemerer, dependencies among classes can be measured by afferent and efferent coupling. Afferent coupling (Ca) computes the number of classes from other packages that depend on classes in the analyzed package (incoming dependencies). Efferent coupling (Ce) counts the number of types of the analyzed package depending on other package (outgoing dependencies) [2]. However, Ca and Ce metrics analyze dependencies at the package level and cannot be used to locate Feature Envy code smell because this code smell is the consequences of too many method dependencies between classes.

The proposed metric is defined based on the idea that methods defined in the same class must be highly cohesive i.e. actively interact with other members in the same class while having less interactions with other classes. In this context, interactions are identified by calls made to the method internally and externally. Internal interactions are calls to methods defined in the current class while external interactions are calls made to methods defined in other classes. The aim is to check whether external interactions are stronger than internal interactions. More external interactions imply that the method is coupled with other classes.

We define the call set (*CS*) as the set of calls made on the given object *obj* inside the method *mtd* as shown in (1).

$$CS(obj, mtd) = \{\, m \mid m \text{ is a method call on } obj \text{ within } mtd \,\} \tag{1}$$

After the call set is determined, the Feature Envy factor is then computed for a given the object *obj* and a method *mtd* as follows:

$$FEF(obj, mtd) = w(m / n) + (1 - w)(1 - x^{m}) \tag{2}$$

where *m* is the number of calls on the object *obj* i.e. the cardinality of the call set, *n* is the total number of calls on any objects defined or visible in the method *mtd*, *w* is the weight and *x* is the base and $0 \le w, x \le 1$.

Let's break down the calculation of Feature Envy Factor in equation (2). The first term of the equation determines how frequent *obj*'s methods are used with respect to the total calls in the method. The second term determines the severity of external interactions. The values computed by our metric will fall into the range of [0, 1]. Having the values bounded in the range makes it easy to compare different factors.

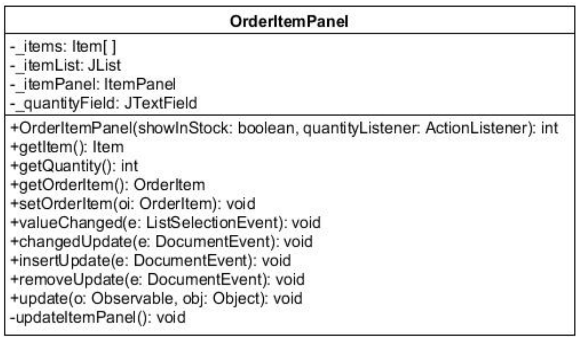
OrderItemPanel
-_items: Item[ ]
-_itemList: JList
-_itemPanel: ItemPanel
-_quantityField: JTextField
+OrderItemPanel(showInStock: boolean, quantityListener: ActionListener): int
+getItem(): Item
+getQuantity(): int
+getOrderItem(): OrderItem
+setOrderItem(oi: OrderItem): void
+valueChanged(e: ListSelectionEvent): void
+changedUpdate(e: DocumentEvent): void
+insertUpdate(e: DocumentEvent): void
+removeUpdate(e: DocumentEvent): void
+update(o: Observable, obj: Object): void
-updateItemPanel(): void

Figure 1. Class diagram of the `OrderItemPanel` class.

To illustrate how the metric works, consider the class diagram of the `Item` class (Figure 1) and the implementation of the `updateItemPanel()` method (Figure 2). With general observation, the method is short and concise. However, with a closer look, we will see that method `updateItemPanel()` sends a lot of messages to `_itemPanel` which is a field of the class.

```
    private void updateItemPanel() {
        Item item = getItem();
        int q = getQuantity();
        if (item == null) {
            _itemPanel.clear();
        } else {
            _itemPanel.setItem(item);
            int inStock = Warehouse.getInstance()
                                .getQuantity(item);
            _itemPanel.setInstock(q <= inStock
                                && 0 < inStock);
        }
    }
```

Figure 2. Feature Envy candidate.

Our approach in detecting Feature Envy in the `updateItemPanel()` method starts from locating all fields, parameters and locally defined objects that are referenced in the method. There are two objects in this case which are `item` (local object) and `_itemPanel` (field). We then determine the call set and the Feature Envy factor for these two objects as follows:

$$CS(\texttt{item}, \texttt{updateItemPanel}) = \{\ \}$$

$$m = |\ CS(\texttt{item}, \texttt{updateItemPanel})\ | = 0$$

$$n = 6$$

$$FEF(\texttt{item}, \texttt{updateItemPanel}) = 0.5 * (0/6) + 0.5 * (1 - 0.5^0) = 0$$

$$CS(\texttt{\_itemPanel}, \texttt{updateItemPanel}) = \{\texttt{clear}, \texttt{setItem}, \texttt{setInStock}\}$$

$$m = |\ CS(\texttt{\_itemPanel}, \texttt{updateItemPanel})\ | = 3$$

$$n = 6$$

$$FEF(\texttt{\_itemPanel}, \texttt{updateItemPanel}) = 0.5 * (3/6) + 0.5 * (1 - 0.5^3) = 0.6875$$

The Feature Envy factor of `item` object is 0 while the `_itemPanel` field gets the value of 0.6875. Our current implementation uses $w = 0.5$ and $x = 0.5$.

B. Metric Validation

The proposed metric is validated based on Henderson-Sellers' set of desirable properties of software metrics [6]. Each property is discussed in detail below.

- Validity is the ability to measure what we intend to measure. Internal validity refers to how well a measure captures the meaning of things or concepts that we want to measure. External validity refers to the generalization. The proposed feature envy metric possesses both the internal and external validity. In regards to internal validity, the metric is able to identify Feature Envy. On external validity issue, the metric can be generalized and is applicable to other object-oriented programming languages even though it was first designed for Java code.
- Reliability is the ability to provide consistent measurements. The degree of reliability is high if repeated measurements taken on the same subjects are highly consistent or even identical. The metric is reliable because it always gives the same result for the same set of inputs and parameters.
- Robustness is the ability to handle and incomplete information and incorrect input. The only input to the metric is the source code that is free of compile-errors or the intermediate form of the software system. It does not have to be the complete system. robustness of this metric,

IV. FEATURE ENVY REMOVAL

Detecting code smells is just a half way towards the goal of improving system maintainability as it only points out the problems. To complete the whole cycle, detected code smells must be removed and this process can be done through refactorings. Refactorings, as defined by William Opdyke, are behavior-preserving program transformations [12]. They involve code restructuring without changing the behavior of the system. As code smells are signs of design flaws or inappropriate system structure, they can usually be removed by restructuring the code. The main focus of such restructuring is to maintain the system's semantics.

Feature Envy code smell once identified can be removed by applying 1) Move Method refactoring 2) Extract Method followed by Move Method refactoring and 3) Move Field refactoring [4]. This work considers only Feature Envy instances that can be removed by Move Method refactoring. In order for the move to take place, all callers of the old method must be identified and all references must be updated. The Move Method refactoring involves 3 parameters:

- a method to be moved
- the defining class
- the target class

Though the concept of moving a method is simple, performing the actual move in the source code is not quite an easy task when system behaviors have to remain unchanged. The process can be error-prone if done by hand. The complexities of move method refactoring involve checking the pre-conditions to ensure that the semantics of the system remain intact if the method has been moved.

A. Method Name Conflict

The first pre-condition that we need to check is whether the target class contains a method with the same name as the one to be moved from the defining class. This condition seems very simple; however, without checking such a pre-condition before performing the method move, two scenarios could occur:

1. If the signatures of the method are the same, there will be compilation errors because the method is redefined

2. If the signatures of the method are different, method overloading will be introduced.

To better illustrate the scenario, see Figure 3. Suppose we want to move the method `getRemainingLife()` from the `Tire` class to the `WarrantyInformation` class which has already defined a `getRemainingLife()` method (with a different method signature), the move will result in the original method being overloaded. Even though the external behavior may be untouched, the software design is altered and is different from the original design intent. This issue can be easily resolved by renaming the method before the move.

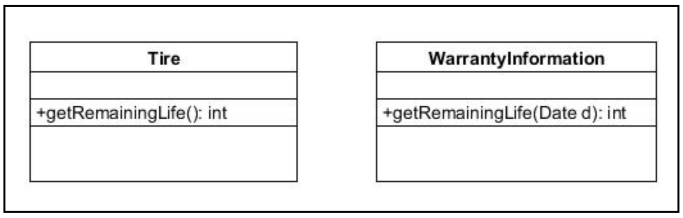

Figure 3. Method name conflict.

B. Overriding Method

If the method to be moved overrides its superclass method, moving the overriding elsewhere will definitely change the system behavior. It does not matter whether the target class is in the same or different inheritance hierarchy. All calls to the overriding methods will get the behavior defined in the parent class instead. As shown in Figure 4, if the `print()` method in the `FarewellMessage` class is moved to some other class, the main method will print two lines of "Hello". While the original program prints "Hello" on the first line and "Good Bye" on the second line.

The conservative approach is to prevent moving any overriding methods. However this restriction can be loosened by applying Extract Method refactoring first so the message in the overriding method is delegated to the newly extracted method. After the method extraction, the extracted method can then be moved to the target class. Our current implementation is taking the conservative approach.

C. Aggregation

One important issue to consider when moving a method concerns the aggregation relationship. Aggregation is also known as a part-of relationship. The concept of aggregation implies that the part can only belong to one whole at a time.

Additional complication comes from the fact that each part in one whole must be unique. As shown in Figure 5, the four tires i.e. `leftFrontTire`, `rightFrontTire`, `leftRearTire` and `rightRearTire` are parts of an `Automobile`. Each tire must be unique, in other words, `leftFrontTire` has to be different from `rightFrontTire`. They must be different objects.

Moving a method that refers to unique components without performing such precondition checks generates the possibility of inadvertently changing behavior. Tsantalis and Chatzigeorgiou [17] suggest that the relationship between the source and the target class must be one-to-one. However, from our viewpoint, the move method refactoring can be made even if the relationship is aggregation or composition. This issue can be resolved by incorporating a proper program analysis with the pre-condition checks. Uniqueness analysis [1] is required in this case.

```
class Message {
    public void print() {
        System.out.println("Hello");
    }
}

class FarewellMessage extends Message {
    public void print() {
        System.out.println("Good Bye");
    }
}

class TestMessage {
    public static void main(String[] arg) {
        Message m = new Message();
        Message n = new FarewellMessage();

        m.print();
        n.print();
    }
}
```

Figure 4. Overriding Method.

```
public class Automobile {
    Engine autoEngine;
    int numOfPassenger;
    Tire leftFrontTire;
    Tire rightFrontTire;
    Tire leftRearTire;
    Tire rightRearTire;
    ...
}
```

Figure 5. Aggregate/Component Relationship.

V. EXPERIMENTAL RESULTS

The tool JCodeCanine is developed as an eclipse plug-in in order to validate our metric. The test is performed on projects developed by students who have limited knowledge on object-oriented concepts. The smallest project consists of 8 classes with 558 LOC while the largest project contains 46 classes with 2,632 LOC. In this section, we look at the accuracy of Feature Envy detection and how Feature Envy removal affects the software qualities.

Our initial results indicate that 62% of Feature Envy detected is false positives. Further investigation reveals that one case of false positives is delegation. However, delegation is a valid approach in object-oriented design and should not be identified as Feature Envy. After revising the implementation and adjusting some parameters, the number of false positives has dropped to 34%.

Another case of the false positives that are detected during the evaluation is illustrated in Figure 6. Based on our metric, *FEF*(`_order`, `setOrder`) = 0.7083 which is a good Feature Envy candidate. However, when looking at line 5 of the method, the `_order` object is redefined. The `_order` object before and after line 5 could be different objects. Moving the method to the target class may change the semantics of the system.

According to the results, it is evident that using the metric alone is insufficient. Program analysis must be incorporated in order to improve the detection accuracy.

```
public void setOrder(Order o) {
   if (_order != null) {
     _order.deleteObserver(this);
   }
   _order = o;
   _order.addObserver(this);
   update(o, o);
}
```

Figure 6. False positive.

Since the main objective of detecting and removing code smells is to improve coupling and cohesiveness. Table I shows resulting measures of each package in the largest student project. Cohesion, coupling and complexity measures are considered in this evaluation. LCOM is used for cohesion measures, afferent coupling (Ca) and efferent coupling (Ce) metrics are used for coupling measures and for system's overall complexity, we use nested block depth (NBD), weighted method per class (WMC) and McCabe's cyclomatic complexity (MC) [2, 6, 9].

TABLE I. SOFTWARE QUALITY MEASURES

Package	LCOM	Ca	Ce	NBD	WMC	MC
`default`	0.250	0	1	1.267	10.000	2.000
`inventory`	0.000	10	1	1.533	16.000	2.133
`inventory.gui`	0.163	4	5	1.363	17.750	1.560
`inventory.xml`	0.396	1	8	1.562	10.500	1.726
`inventory.net`	0.277	2	2	1.707	7.100	1.732
`io`	0.068	5	7	1.281	5.545	1.906

As illustrated in Figure 7, we compare the software qualities before and after applying suggested refactorings. Our observation reveals that most of the suggestions are in `inventory.gui` package. The results show that there is slight improvement on the NBD and McCabe's complexity metric while WMC measure is decreased. Other cohesion and coupling metrics show no improvement.

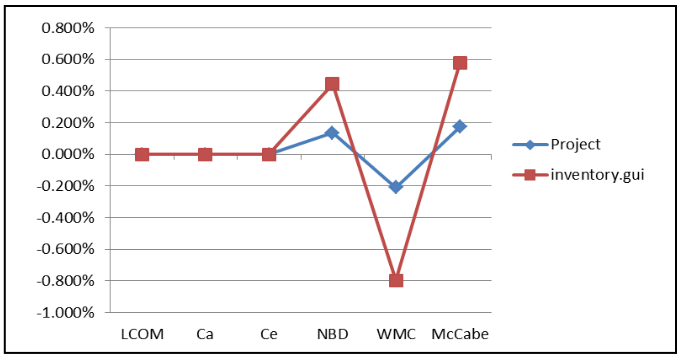

Figure 7. Sofware quality comparison.

Table II compares the number of Feature Envy instances detected by our tool, JCodeCanine and those that are detected by JDeodorant (version 5.0.22 as an eclipse plug-in) [17]. The figures show that out of all instances detected, 80% of them are commonly detected by both tools. Such results validate our approach's ability to identify Feature Envy. Furthermore, our tool is more flexible than JDeodorant as it allows users to adjust the parameters (*w* and *x*) and the threshold as they see appropriate.

The current JDeodorant only detects Feature Envy and the developers have to rely on eclipse's refactoring feature for moving the method. JCodeCanine, on the other hand, offers both Feature Envy detection and removal. It is a semiautomatic tool. After Feature Envy is detected, the developer has an option whether to proceed with the recommended method move. JCodeCanine will then perform the move upon the user's request.

TABLE II. NUMBERS OF FEATURE ENVY DETECTED

Package	JCodeCanine	JDeodorant	JCodeCanine & JDeodorant
`default`	1	0	0
`inventory`	3	3	3
`inventory.gui`	5	6	5
`inventory.xml`	1	0	0
`inventory.net`	0	0	0
`io`	0	0	0

VI. CONCLUSION AND FUTURE WORK

The design of object-oriented systems should be loosely coupling and high cohesion. Placing features in the wrong class is improper relationship among classes in object-oriented systems. A new metric to detect Feature Envy code smell is proposed. The results are promising as they show some improvements on the quality of the tested systems. Feature Envy Factor can be applied to systems developed using any object-oriented programming languages.

It is also worth noting that while other related works focused only detecting Feature Envy or identifying Move Method refactoring candidates and rely on other refactoring tools for moving the methods, this work looks at the complete process which is Feature Envy detection and removal.

Our main focus for future research is to reduce the number of false positives by incorporating analysis with the metric that checks and makes sure that the object in question is not redefined inside the method. Further investigation must also be done on false negatives or Feature Envy that is not detected. We plan to validate our metric on large opensource systems. However, one of the challenges in determining the accuracy of feature envy detection on opensource projects is that we need to understand developers' design intents. We intend to adopt the concept of gold set [3] in future evaluations.

One important issue in our agenda is to explore the impact of parameter values. Since different types and/or domains of applications may require different weights and thresholds for the metric, it is plausible to adopt a machine learning technique that could make the tool adaptive with respect to the metric parameters and the threshold value [7].

ACKNOWLEDGMENT

The author would like to thank anonymous reviewers for valuable comments and suggestions.

REFERENCES

[1] J. Boyland, “Alias burying: Unique variables without destructive reads,” Journal of Software Practice and Experience, vol. 31, no. 6, pp. 533–553, May 2001.

[2] S. R. Chidamber and C. F. Kemerer, “A metric suite for object oriented design,” IEEE Transactions on Software Engineering, vol. 20 iss. 6, pp. 476-493, 1994.

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: a taxonomy and survey,” Journal of Software: Evolution and Process, pp. 53-95, 2013.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, “Refactoring: Improving the design of existing code,” Addison Wesley Longman Reading, Massachusetts, USA, 1999.

[5] G. Gui and P. D. Scott, “Measuring software component reusability by coupling and cohesion metrics,” Journal of Computers, vol. 4, no. 9, pp. 797-805, September 2009.

[6] B. Henderson-Seller, “Object-Oriented Metrics: Measures of Complexity,” Prentice Hall, 1996.

[7] J. Kreimer, “Adaptive detection of design flaws,” Electronic Notes in Theoretical Computer Science, vol. 114, iss. 4, pp. 117-136, 2005.

[8] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information", Proceedings of 23rd International Conference on Software Engineering, pp. 103-112, 2001.

[9] T.J. McCabe, “A complexity measure,” IEEE Transaction on Software Engineering, vol. 2, no.4, pp. 308–320, 1976.

[10] T. M. Meyers and D. Binkley, "Slice-based cohesion metrics and software intervention", Proceedings of 11th Working Conference on Reverse Engineering (WCRE’04), pp. 256-265, 2004.

[11] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk and A. De Lucia, “Identifying method friendships to remove the feature envy bad smell,” Proceedings of the 33rd International Conference on Software Engineering, pp. 820–823, 2011.

[12] William Opdyke, “Refactoring Object-Oriented Frameworks,” PhD thesis, Univer sity of Illinois, 1992.

[13] L.M. Ott and J.J. Thuss, "Slice based metrics for estimating cohesion," Proceedings of the 1st International Software Metrics Symposium, pp.71-81, 1993.

[14] S. Patel, W. Chu, and R. Baxter, "A Measure For Composite Module Cohesion", Proceedings of International Conference on Software Engineering (ICSE'92), pp. 38-48, 1992.

[15] V. Sales, R. Terra, L. F. Miranda and M. T. Valente, “Recommending move method refactorings using dependency sets,” Proceeding of the 20th Working Conference on Reverse Engineering (WCRE’13), pp. 323-241, 2013.

[16] R. Shatnawi and W. Li, “An empirical assessment of refactoring impact on software quality using a hierarchical quality model,” International Journal of Software Engineering and Its Applications, vol. 5, no. 4, pp. 128-149, October 2011.

[17] N. Tsantalis and A. Chatzigeorgiuo, “Identification of Move Method Refactoring Opportunities,” IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347-367, 2009.

[18] N. Tsantalis and A. Chatzigeorgiuo, “Ranking refactoring suggestions based on historical volatility,” Proceedings of the 15th European Conference on Software Maintenance and Reengineering, pp. 25-34, 2011.

[19] A. Yamashita and L. Moonen, “To what extent can maintenance problems be predicted by code smell detection? – An empirical study,” Information and Software Technology Journal, vol.55, iss. 12, pp. 2223-2247, 2013.