Two Level Dynamic Approach for Feature Envy Detection

Swati Kumar
Department of Computer Engineering
National Institute of Technology, Kurukshetra
Kurukshetra, India
swati1222k@gmail.com

Jitender Kumar Chhabra
Department of Computer Engineering
National Institute of Technology, Kurukshetra
Kurukshetra, India
jitenderchhabra@rediffmail.com

***Abstract*—Refactoring leads to more maintainable software. To refactor the code, it must be known which part of code needs to be refactored. For this purpose code smells are used. Detecting code smells in itself is a challenging task. In this paper we propose a technique based on dynamic analysis for the detection of Feature Envy code smell. Feature envy is a method level smell and occurs when a method is more interested in another class than its own class. Previous approaches detects feature envy by analyzing each method in the system, which incurs an overhead as those methods which are closely bounded to their classes are included in the detection mechanism. To tackle this problem, we have devised a two level mechanism. First level filters out the methods which are suspects of being feature envious and second level analyzes those suspects to identify the actual culprits. The proposed technique has been evaluated and compared with existing techniques.**

Keywords*—code smell; dynamic analysis; software maintenance; object –oriented programming

I. Introduction

Nowadays, software maintenance is getting attention of researchers, as the maintenance evolution costs of software projects are significantly larger than the cost of initial development [2]. A software system becomes harder to maintain as it evolves over a period of time. Its design becomes more complicated and difficult to understand; hence it is necessary to reorganize the code once in a while. The most important thing when reorganizing code is to make sure that the program behaves the same way as it did before the reorganization has taken place. This is ensured by performing refactoring. Refactoring term was coined by W. Opdyke [5]. According to Fowler et al. [1] refactoring is defined as-"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." To automate the refactoring, it must be find out where to apply refactoring. For this purpose code smells were introduced by Fowler.

Fowler et al. [1] gave informal description of 22 code smells as indicators of the code which need to be refactored. Bad smells answers the question- what and when to refactor. Duplicated code, Long method, large class, shotgun surgery and long parameter list are some examples of code smells. List of smells is not limited to only these 22 smells. Various other kinds of bad smells have been described in other publications [4], [11].

In object oriented programing it is a key point that data and the functions using that data should be packaged together. Feature envy denotes the violation of this principle. It occurs when a method seems to be more interested in some other class than the one it is defined in [1]. The method invokes much more methods or access fields from another class than its own class. A feature envious method is shown in Figure 1 findEntry() method is calling repeatedly methods from class SearchDialog. Feature envy introduces improper coupling between classes and low cohesion within the class. According to good design principles and laws of object oriented programming, a system should have low coupling and high cohesion [6].

```
boolean findEntry(SearchDialog searchDialog)
{

Cursor waitCursor = shell.getDisplay().getSystemCursor
(SWT.CURSOR_WAIT);
shell.setCursor(waitCursor);

boolean matchCase = searchDialog.getMatchCase();
boolean matchWord = searchDialog.getMatchWord();
String searchString = searchDialog.getSearchString();
int column = searchDialog.getSelectedSearchArea();
```

Fig. 1. Example of Feature Envy.

Today detection of code smells is a challenging task in the industry. Several approaches, as detailed in Section II, have been proposed to detect smells. They have two limitations- first, they analyse the source code statically, which is less accurate in object oriented environment [7]. Second, the detection mechanism is applied over complete code blindly, irrespective of whether they are completely free of any kind of smell or not, which is an overhead.

Dynamic analysis is a technique in which data is obtained by execution of the program instead of only scanning the source code. Therefore, the data obtained is the actual behavior of the code. In static analysis we only assume what may happen when code is executed. Hence, dynamic analysis is more accurate in object oriented environment.

In this paper we propose detection of feature envy smell using dynamic analysis. This approach is two phased- first those methods are extracted which can be possibly feature envious, based on the number of methods called by them from different classes than the one they belong. Then, these suspects are analysed further to confirm whether they are feature envious or not. To confirm, it is checked that majority of variables accessed and methods called are from a single class. We have evaluated our approach on packages from open source system and compared our results with JDeodorant [22] and inCode [23].

Rest of the paper is organized as follows: Section II surveys related work. Section III describes the proposed approach. Section IV presents case study. Section V concludes the paper.

II. RELATED WORK

M. Fowler et al. [1] introduced the term bad smell and enlisted 22 smells and suggested how to use them for refactoring. M□ntyl□ [3] proposed classification of 22 bad smells based on their common characteristics. Wake [4] classified bad smells based on the location- within a class and between classes.

Manual detection of bad smells is a time-consuming and error-prone activity. To automate the detection procedure several studies have been done in past decade.

Emden and Moonen in [8] categorized code smells into two kinds of aspects: primitive smell aspects and derived smell aspects. Primitive smell aspects can be observed directly from the code and derived smell aspects are inferred from other aspects according to a set of inference rules. They suggested that code smells are subjective, domain-dependent and not precise in terms of underlying parameters. They developed jCosmo in which addition or removal or changes in definition of smells can be done easily. They also suggested that some code smells need runtime information and thus dynamic analysis can be used. But, such code smells were not pointed out.

Marinescu [9] proposed a metric-based approach to detect code smells called "Detection Strategies". The strategies capture deviations from good design principles. Detection Strategies consist of metrics values combined together with set operators. Each metric value is associated with a threshold. Threshold can be relative or absolute. Explanation for choice of metrics and threshold values is missing.

Munro [10] suggested that text based definition of smells for purpose of their detection is not appropriate, as interpretation of definition varies from person to person. He gave more precise specification of bad smells in form of a template to describe smells formally. It consists of three main parts: a code smell name, a text-based description of its characteristics, and heuristics for its detection. He also proposed metric based heuristics to detect code smells. He gave justification for the choice of metrics and thresholds for detecting smells and empirically validated the approach.

M. Salehie et al. in their work [12] proposed that a gap exists between metrics used for detection purpose, design rules and quality factors. A framework is designed which quantify design flaws at the class level and relate them to different features (i.e. complexity). Proposed approach focus only on maintainability. Framework consists of three components: A generic Object Oriented design knowledge-base which has design heuristics, metrics, flaws and their relationships; A hot spot indicator pointing out the most probable defective entities namely "hot spots" using primitive classifiers (set of metrics); A design flaw detector locating possible design flaws in the predetermined hot spots using composite classifiers (set of metrics). Authors also studied relation between smells.

Moha et al. in their studies [13, 14, 15], addressed the issue of lack of formal descriptions of code smells. Their aim was to find a way to formally describe them. They proposed domain-specific language for specifying code smells and automatically generating detection algorithms based on these specifications. This language provides a higher level of abstraction and smell can be added and modified as per the needs.

In a recent study by M. Kessentini et al. [16] detection of code smells is considered as search problem. Authors suggested that manual selection of metrics and forming rules for detection of each and every smell requires effort. They proposed approach for automatic generation of smell detection rules. This approach uses knowledge from previously manually inspected projects in order to detect design defects, called defects examples, to generate new detection rules based on a combinations of software quality metrics. Results showed that their approach is better than DÉCOR [15].

In another recent study [17], Ligu et al. considered Refused Bequest smell which has received limited attention in past. Their approach for identification of Refused Bequest consists of evaluating code statically and dynamically. Static analysis gives suspects and dynamic unit test execution determines subclasses that actually exhibit the smell. This work is an example that initiative have been taken in direction of using dynamic analysis for code smell detection.

For detecting feature envy, rules based on object oriented design metrics are proposed in [9], [10] and [12]. Mainly two issues exists while using metrics for detection. First is deciding threshold values for metrics. As many metrics exist in literature, each metric measures property differently. So, deciding threshold for each and every metric is not feasible. To the best of our knowledge, generalised threshold values have not been suggested. Second, many design metrics have been developed but it is not an easy task to decide which one is suitable and can efficiently detect code smells. Because of this limitations related with metrics two level approach proposed in [12] have limitations for detection of feature envy.

In [22] Tsantalis and Chatzigeorgiou proposed distance based approach for detection and correction of feature envy. But they only considered those feature envious methods which can be refactored using move method refactoring only. The approach measures the distance between system entities (attributes/methods) and classes and extracts a list of behavior-preserving refactorings based on the examination of a set of precondition. They also defined Entity Placement metric that quantifies how well entities have been placed in system classes and measures the effect of all refactoring suggestions. The proposed methodology was also implemented as JDeodorant Eclipse plug-in.

In [18] Dexun et al. proposed weight based distance metrics theory for detecting feature envy. They calculated distance between entities (attributes and methods) and classes. The weight based distance metric considers the multiple invoking relationships between each two entities. This in result reduced false positives and false negatives. They compared their results with simple distance based approach.

III. PROPOSED WORK

This paper proposes a two level approach for detection of feature envy code smell using dynamic analysis, as described in Figure 2 below. The proposed approach uses dynamic analysis at both levels to increase the accuracy of detection by considering the actual execution performance instead of the static behaviour. At first level, suspects are obtained dynamically. Suspects are methods which may possibly suffer from feature envy. At second level, it is confirmed whether a suspect is feature envious or not. Elimination of non-suspect methods reduces overhead and make the detection process efficient.

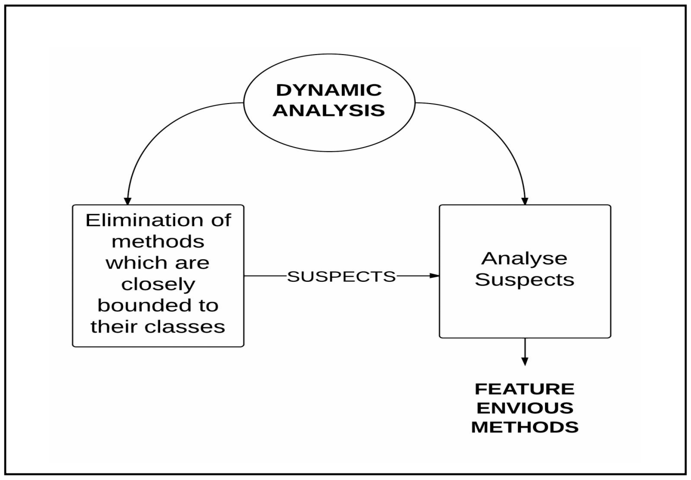
DYNAMIC ANALYSIS
Elimination of methods which are closely bounded to their classes
SUSPECTS
Analyse Suspects
FEATURE ENVIOUS METHODS

Fig. 2. Overview of proposed approach

Previous approaches detected feature envy by statically analysing source code. However, these approaches, did not consider how program elements cooperate with one another in program execution, so that those might miss some bad smells or might detect some false. Consider the method in Figure 3. When jDeodorant was used to detect whether myMethod is feature envious or not, jDeodorant detected it as feature envious. In first look it seem obvious that myMethod is feature envious of Myclass. But when myMethod is executed, it is found that myMethod did not use variables of Myclass. Clearly, myMethod is not feature envious, but a dead code. But this can be found only when code is executed. Hence, static analysis has limitations.

```
1. Public void myMethod (MyClass object)
2.     for (int i=0; i<10;i++)
3.         if ( i==11)
4.             object.num1=10;
5.             object.num2=20;
6.               object.num3=30;
```

Fig. 3. A case of dead code which is detected as feature envy by static approach but not by dynamic approach.

As previous approaches analyse each and every method present in the system, which is an overhead. So, the proposed approach being two level reduces this overhead. It has one more benefit. It is a well-known fact that dynamic analysis is expensive. This is one of the reasons why it is not explored for detection of code smells as much as static analysis has been used. Hence, reducing number of methods to be analysed compensate the cost and complexity of dynamic analysis.

Next we are going to discuss each step in detail. Figure 4 shows working of two levels.

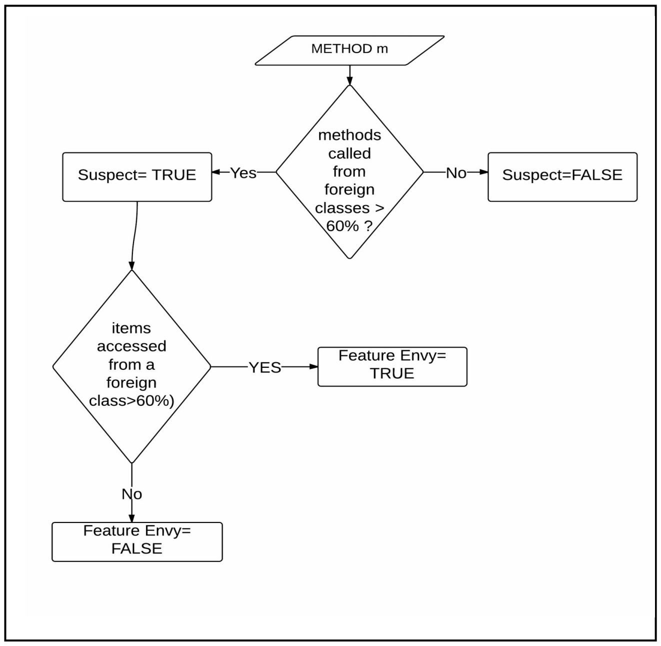
METHOD m
methods called from foreign classes > 60% ?
Yes
No
Suspect= TRUE
Suspect=FALSE
items accessed from a foreign class>60%)
YES
Feature Envy= TRUE
No
Feature Envy= FALSE

Fig. 4. Working of feature envy detection mechanism

(Step 1) ***Obtain suspects***: First of all the program is executed in a way that it is ensured to cover the whole. This ensures that no method is left out. Here only those methods are selected as suspects which make a large number of calls to methods belonging to classes different from the class of measured method. A threshold of 60% (lower bound) is decided. This threshold is decided to get a balance between covering all opportunities of detection and eliminating non-suspects. If threshold is lowered then those methods can also

get selected as suspect, which are closely bounded to their class. If we increase it then there are chances to miss some opportunities.

For this step we have used jProfiler [24]. It is a profiler which analyze java code dynamically. It attaches with Java Virtual Machine (jvm) and measures the execution paths of the application on the method level. Information for method calls is extracted using jProfiler.

(Step 2) ***Confirmation***: From the suspects, now we decide which method are actually feature envious. For this, the behavior of the suspect is analysed by executing the program and capturing only details of suspect methods only. Details include number of variables accessed and method calls made. A method is declared feature envious if it is accessing items (methods and attributes) of a particular class more than any other class.

If 60% (lower bound) of items are accessed of one particular class, then method is feature envious to that class. 60% is chosen as threshold because in a system a method interacts with many other methods. So, setting a high threshold will only detect those feature envious methods that can be completely moved. Setting this threshold will make sure that most of the feature envious methods are detected, especially those methods in which only some part is feature envious. If we increase the threshold more then there are chances that such opportunity will be missed.

Methods which are calling only one method and that too of some other class and also no attributes are accessed are delegate methods. Delegate methods simply pass the responsibility to some other method. Delegate methods are not considered as feature envious. Through them methods are calling other class's method. To make sure that actual method accessed is included in a method's detail, we replace the delegate method's call with the actual method being called and methods are analysed again. As delegate methods are very less as compared to methods providing some functionality, they are considered in later stage.

For this step we have used AspectJ [21], which is an Aspect Oriented Programming (AOP) [20] extension. Dynamic analysis is less popular and costly mainly due to the technical difficulties associated with the data collection. Data is collected by inserting some points for instrumentation in the code. By using AOP, the code can be instrumented automatically by inserting data collection points to the source code or even to the byte code. This significantly reduces the effort needed for code instrumentation compared to manual instrumentation. AspectJ has become a widely used language for AOP because of its simplicity and usability for end users. It uses Java-like syntax. It has following constructs: pointcuts, advice, inter-type declarations an aspects.

A *join point* is a well-defined point in the program flow, example is a method call. A *pointcut* picks out certain join points and values at those points. A piece of *advice* is code that is executed when a join point is reached. These are the dynamic parts of AspectJ. AspectJ also has different kinds of *inter-type declarations* that allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes. AspectJ's *aspect* are the unit which constitute above three constructs. It is like a class. AspectJ collects data by aspect *weaving*, which composes the code of the base classes and the aspects to ensure that applicable advice runs at the appropriate join points.

Data collected during program execution using AspectJ is shown in Figure 5 below. topIndex and index are variables with details of the class to which they belong. Methods called by a method are shown in line starting with call. In this way we collected details of a method using AspectJ.

topIndex
class:org.eclipse.swt.examples.accessibility.CTable
index
class: org.eclipse.swt.examples.accessibility.CTableItem

call (int org.eclipse.swt.examples.accessibility.
CTableItem. getTextX(int))
call (int[]
org.eclipse.swt.examples.accessibility.CTable.getColumn
Order())

Fig. 5. Data collected during program executon using AspectJ.

In both steps, method calls and accessing attributes of superclass must be excluded i.e. if an item accessed by a method belongs to superclass of the class in which method is defined, should not be considered foreign.

IV. Case Study

A. Experimental Setup

We have tested our approach by detecting feature envious methods in Eclipse's example for the Standard Widget Toolkit (SWT) [1] and one package from jEdit 3-5.1.0 [2], latest version. jEdit is a text editor for programmers, available under the GNU General Public License version 2.0. It is written in Java and runs on any operating system with Java support. We have chosen package textarea due to its large number of classes and large methods per class.

Size properties are shown in Table I and Table II for SWT example and textarea respectively. Main method and methods in interface are excluded from total number of methods in the system. We have compared our results with JDeodorant [3] and inCode [4].

B. Results

13 methods in SWT examples and 12 methods in textarea were detected as feature envious.

1)SWT Examples: Out of 16 packages only 2 packages were found to be affected by feature envy. While suspects were from 3 packages only. Out of 26 suspects only 13 were found to be feature envious. 98.40% reduction in effort was

1 http://www.eclipse.org/swt/examples.php
2 http://sourceforge.net/projects/jedit/
3 http://www.jdeodorant.com/
4 http://www.intooitus.com/products/incode

observed.

TABLE I. SIZE PROPERTIES OF SWT EXAMPLE

Lines of Code	Number of Packages	Number of Classes	Number of Methods	Method/Class
45947	16	148	1665	4.05

TABLE II. SIZE PROPERTIES TEXTAREA PACKAGE

Lines of Code	Number of Classes	Number of Methods	Method/Class
11509	43	755	11.65

inCode detected 2 methods as feature envious. Our approach found only one of them suffering from feature envy. We have verified manually that the method which was not found to be feature envious by our method is in actual not feature envious. But is detected by inCode because it do not consider method dependencies and inheritance. While JDeodorant detected 13 methods as feature envious. One is not detected by our approach.

It was found that 80% of the feature envious methods belonged to one class. According to Pietrzak and Walter [19] the already confirmed presence or absence of a particular smell may carry information about others. According to them feature envy supports large class. So, there may be chances that this class can be large class. It needs more investigation, which is out of scope of our work.

2) *Package textarea*: 12 methods were found to be feature envious out of 54 suspects. Out of 43 classes, feature envious methods belonged to 6 classes. Class JEdittextArea inherits from TextArea class. The function smartHome() was detected as feature envious when inheritance was not taken into account. Hence inheritance should always be considered while detecting feature envy.

inCode detected 8 methods as feature envious, out of which only 1 was detected by our approach, 1 is not detected and other 6 are eliminated. As inCode do not consider method dependencies, methods detected by it as feature envious are not in actual. JDeodorant detected 5 methods. Out of 5, one is eliminated and another one is not detected by our approach.

As observed from Table III, results show that proposed approach can systematically and efficiently detect design flaws of an object oriented java based systems. Our results agree with JDeodorant. But we have considered detection of feature envy which is not refactoring specific. Hence, our approach is more generalized. As it is clear from the results that the flawed methods are very less as compared to total methods present in the system. Hence, a lot of effort is saved, making the detection mechanism more efficient.

C. Threats to validity

In the evaluation, we have tried to cover the whole code. However, method invocations depends on the user scenario; thus, we obtain different method invocations by running different combinations of commands. Additionally, the number of method invocations affects the detection of feature envy. Thus, the precision of detection mechanism depends on what functionalities of system were used and under what conditions.

TABLE III. RESULTS

System	Reduction in Effort	Accuracy Improved w.r.t inCode	Accuracy Improved w.r.t JDeodorant
SWT Example	98.40	50.0%	0%
Textarea	92.84	75.0%	20%

V. CONCLUSION

Code smell detection is an extremely challenging maintenance task. In this paper the use of dynamic analysis for detection of bad smells is explored. We have attempted to confront the problem of diagnosing Feature Envy smell, employing a two phased dynamic approach for object oriented Java systems. Our results proved that elimination of flaw free methods before detection saves a lot of effort.

We have evaluated our approach on two open source systems. The results obtained through our approach are more accurate and obtained in lesser time. Our approach was able to detect proper Feature Envy code smell. On the basis of our study, it can be concluded that use of dynamic analysis can be advantageous in detection of other types of code smells also and will be a useful and efficient approach for software maintainers.

REFERENCES

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

[2] A. Yamashita, and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection? – An empirical study," *Information and Software Technology*, vol. 55, no.12, pp. 2223–2242, 2013.

[3] M.V. M□ntyl□, and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Journal of Empirical Software Engineering*, vol. 11, no. 3, pp. 395-431, 2006.

[4] W.C.Wake, *Refactoring Workbook*. Addison-Wesley, 2003.

[5] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD dissertation, Univ. of Illinois at Urbana-Champaign, 1992.

[6] R. S. Pressman, *Software engineering -A practitioner's approach*. 5th ed., McGraw-Hill, 2001.

[7] A. Mitchell, "An empirical study of run-time coupling and cohesion software metrics," Ph.D. dissertation, National Univ. Ireland, Ireland, 2005.

[8] E. van Emden, and L. Moonen, "Java quality assurance by detecting code smells," *Proc. 9th Working Conf. on Reverse Engineering*, Oct. 2002.

[9] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” *Proc. 20th Int’l Conf. on Software Maintenance*, pp. 350–359, Sept. 2004.

[10] M.J Munro, “Product metrics for automatic identification of “bad smell” design problems in java source-code,” *Proc. 11th Int’l Software Metrics Symposium*, Sept. 2005.

[11] M. Lanza, and R. Marinescu, *Object Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer-Verlag, 2006.

[12] M. Salehie, S. Li, and L. Tahvildari, “A metric-based heuristic framework to detect object-oriented design flaws,” *Proc. 14th Int’l Conf. on Program Comprehension*, pp. 159-168, 2006.

[13] N. Moha, Y.-G. Guéhéneuc, A.-F. L. Meur, and L. Duchien, “A domain analysis to specify design defects and generate detection algorithms,” *Proc. 11th Intl Conf. on Fundamental Approaches to Software Engineering*, Apr. 2008

[14] N. Moha, Y.-G. Guéhéneuc, and P. Leduc, “Automatic generation of detection algorithms for design defects,” *Proc. 21st Conf. on Automated Software Engineering*, pp. 297–300, Sept. 2006.

[15] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Trans. Softw. Eng.*, vol. 36, pp. 20-36, Jan./Feb. 2010

[16] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, “Search-based design defects detection by example,” *Proc. 14th Int’l Conf. on Fundamental Approaches to Software Engineering*, pp. 401-415, Apr. 2011.

[17] E. Ligu, A. Chatzigeorgiou, T. Chaikalis and N. Ygeionomakis, “Identification of refused bequest code smells,” *IEEE Int’l Conf. on Software Maintenance*, Eindhoven, Netherlands, pp. 392 – 395, 22-28 Sept. 2013.

[18] J. Dexun, M. Peijun, S. Xiaohong, and W. Tiantian, “Detecting Bad Smells with Weight Based Distance Metrics Theory,” *IEEE Int’l Conf.* on *Instrumentation, Measurement, Computer, Communication and Control* , Harbin , pp. 299-304, 8-10 Dec. 2012.

[19] B. Pietrzak, and B. Walter, “Leveraging code smell detection with inter-smell relations,” *Proc. 7th Int’l Conf. on Extreme Programming and Agile Processes in Software Engineering*, pp. 75-84, June 2006.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin “Aspect-Oriented Programming,” P*roc. European Conf. on Object-Oriented Programming*, pp. 220-242, June 1997.

[21] G. Kiczale., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.J. Griswold, “An overview of AspectJ,” *Proc. European Conf. on Object-Oriented Programming*, pp. 327–353, June 2001.

[22] N. Tsantalis, and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Trans. Softw. Eng*., vol. 35, pp. 347-367, May/June 2009.

[23] R. Marinescu, G. Ganea, and I. Verebi ,“inCode: Continuous quality assessment and improvement,” *Proc. 14th European Conf. on Software Maintenance and Reengineering*, pp. 274-274, 2010.

[24] http://www.ej-technologies.com/products/jprofiler/overview.html