DOMAIN SPECIFIC INFRASTRUCTURE FOR CODE SMELL DETECTION IN LARGE-SCALE SOFTWARE SYSTEMS

M. A. K. Maduranga[1], D. C. Mahagamage[2], P. I. Madhavi [3], J. A. H. Madushan [4], C. Wijesiriwardana[5]

[1]Faculty of Information Technology, University of Moratuwa, Sri Lanka, Email: kavintha.maduranga@gmail.com
2Faculty of Information Technology, University of Moratuwa, Sri Lanka, Email: dinushamahagamage3403@gmail.com
3Faculty of Information Technology, University of Moratuwa, Sri Lanka, Email: imalkamadhavi@gmail.com
4Faculty of Information Technology, University of Moratuwa, Sri Lanka, Email: harithm918@gmail.com
5Faculty of Information Technology, University of Moratuwa, Sri Lanka, Email: chaman@uom.lk

ABSTRACT

Modern software systems are comprised of thousands of components, built by multiple teams and dozens of developers scattered across geographical boundaries. Ensuring the quality of such software systems requires thorough calibration across multiple technologies, various application domains and clear identification of application boundaries. Therefore, the quality cannot be guaranteed at the level of individual developers or quality assurance engineers, without exploiting sophisticated automated tools. As a result, software analysis; static code analysis in particular has become a gold rush topic among software practitioners as well as software researchers. Despite the benefits of using such static analysis tools to improve software quality (i.e., find bugs, find code clones), recent studies suggest that these tools are underused. Therefore, in this research, we tackled that problem in a different angle by introducing a novel Domain Specific Language (DSL) for static analysis, with a particular focus on detecting bad smells in the source code. In our opinion, the DSL that was developed during this research is a stepping-stone to guide software analysis research to a higher level. In addition to the basic code smells detection; the underline infrastructure allows integrating with version control repositories and various other project management tools to ensure several rich functionalities.

Key words**:** Software Analysis, Code Smells, Domain Specific Languages

1. INTRODUCTION

Quality assurance of large-scale software systems requires thorough calibration across multiple technologies, various application domains and clear identification of application boundaries. As a result, guaranteeing the quality of software projects are cumbersome without exploiting sophisticated automated tools. As a result, software analysis; static code analysis [1] in particular has become a gold rush topic among software practitioners as well as software researchers.

Code smells are considered as structural characteristics of software that could indicate a code or design problem [2]. As a result, code smells have negative impacts on software evolution and maintenance. They are not technically incorrect and do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. Finding Code Smells are one of the most commonly used software analysis method at the design and implementation level. Therefore, many static analysis tools have been developed to detect code smells in software projects. Despite the benefits of using such static analysis tools to improve software quality (i.e., find bugs, find code clones), recent studies suggest that these tools are underused. However, there is a need of a sophisticated mechanism for code smell detection together with rich functionalities such as dynamic detection, linking with web based source code repositories and various other continuous integration tools.

Therefore, in this research, we tackled that problem in a different angle by introducing a new Domain Specific Language (DSL) for static analysis, with a particular focus on detecting bad smells in the source code. DSL is a computer language that intends to solve problems in a particular domain. Not like general-purpose computer languages, DSLs are restricted to a specific domain with specific scope, grammar, vocabulary and senses.

The remainder of this paper is organized as follows. Section 2 presents related work of this research. Methodology in Section 3 followed by Results in Section 4. We conclude with a

Discussion and Future Work in Section 5.

2. RELATED WORK

Code smells; often called as bad smells, design flows, anti-patterns as well as anomalies. Removal of such smells should be done as early as possible to avoid the difficulties in locating them.

2.1. Code Smell Detection

God Class: A God Class performs too much work on its own, delegating only minor details to a set of trivial classes, and using the data from other Classes [3].
Data Class: Classes that passively store data are known as data classes [4]. Basic metrics like number of methods, number of assessors and complexity in a source code are used to detect the data classes.
Long Parameter List: Long parameter list is a parameter list that is too long and thus difficult to understand [3]. Long parameter list was detected by counting the number of parameters of each method with the consideration of the type of parameters.
Duplicated Code: Code duplicates were measured using the percentage of duplicated code lines in the system. Problem with measurement of this smell lies in the different possible kinds of duplication. Even though the exact duplication can be measured, other duplications are much harder to determine. Therefor we used techniques as martin fowler [4] described and it is a research area that grows with respect to time.
Long Method: Long method is a method that is too long, so it is difficult to understand, change or extend [3]. So here we used basic metrics like methods complexity and number of lines of code to detect the long method smell.

A summary of the available code smell detection tools is presented in Table 1.

Table 1: Code Smell Detection Tools and their capabilities

Tool	Type	Analyzed Language	Link to Code
Checkstyle	Eclipse plugin, Standalone	Java	No
Decor	Standalone	Java	No
iPlasma	Standalone	C++, Java	No
infusion	Standalone	C, C++, Java	No
JDeodrant	Eclipse plugin	Java	Yes
PMD	Eclipse plugin, Standalone	Java	Yes
Stench Blossom	Eclipse plugin	Java	Yes

2.2. Overview of Domain Specific Modelling

Domain-Specific Modeling (DSM) has become popular in recent years. This is no surprise because a well-designed DSL can be much easier to program and it improves the programmer's productivity and quality. Nowadays, DSLs have been developed to solve the problems in several domain areas.

The development of DSL for Software Analysis (find code smells in object oriented projects, get issues and other project management details related to projects from project management tools, get commits, branches details related to projects from code repositories) is still not being concerned by the DSL developers to a greater extend.

Boa is one of such languages and it use to mine software repositories. And Boa's infrastructure supports distributed computing techniques to execute queries against hundreds of thousands of software projects [6].

Rascal is another DSL for meta-programming. It supports static code analysis, program transformation and it has libraries for integrating language front-ends and reusing analysis algorithms as well [7]. Likewise domain specific modelling has used in software analysis domain. So it is more important that code smell detection area is yet untouched but has many aspects that needed the guide of domain specific modelling.

Apart from textual languages, several interactive approaches such as mashup like tools have been suggested in the literature to facilitate software analysis [8]. However, such directions need to be further investigated.

3. METHODOLOGY

3.1. Infrastructure Overview

The infrastructure consists of several sub-modules that are interconnected with each other. Below we provide a short description of each module.
(a) Data extractor : The infrastructure is capable

of analyzing both local and remote (e.g., github) projects given the source code is available.

(b) Metrics calculator : Rather than developing our own metrics calculator, we take advantage of Sonarqube to extract software metrics for the projects that are to be analyzed.

(c) User Interface : The UI is provided as an eclipse plug-in where any user can install the plug-in and start using all the features of infrastructure including the DSL coding platform.

(d) Mapper : The mapper acts as an intermediate component to map our DSL scripts to Java source code for the purpose of code smell detection. This contains the logic for analyzing a particular project. The mapper class has been implemented in such a way that all sort of future enhancements as well as modifications are possible with a minimum effort (e.g., integrating Jira).

(e) Grammar of the DSL : It makes the language "live" and giving the knowledge to identify the syntax and semantics of language scripts.

3.2. DSL Overview

The DSL is capable of detecting main code smells in any software system by writing only a minimum number of code lines. In addition to the basic code smells detection; the underline infrastructure of the language allows integrating with version control repositories and various other project management tools to ensure several rich functionalities. The DSL is developed as Eclipse plug-in allowing software practitioners to easily install and analyze software projects.

The "Xtext" [9] framework has been used to develop the DSL easily with existing common libraries, which provide a set of rich features to work with any domain. The "Xtext" framework facilitates the development of domain-specific languages for the Java Virtual Machine, referring to and compiling to Java artifacts with tight integration into Eclipse's Java Development Toolkit. A reusable expression languages library enables rich behavior right within the DSL. Furthermore, Xtext-based languages can be used to drive code generators that target Java, C, and C++, C #, Objective C, Python, or Ruby code. Despite having several click-and-see tools in the industry, the underline principles of this approach is easily extensible to provide a complete software analysis solution in the future.
Xtend was used to translate the grammar of the implemented language to a java source code. This java source code is mapped with the back end implementation using the mapper, which is also written in java. During the processing of the source code, all the code lines will be processed by converting to JVM understandable code and will be compiled under java compiler. So this language depends on JVM and "ANTLR" parser and they will be used within this process.

If someone needs to analyze the quality of a software project, overall quality should be measured not only with source code but also with content management aspects as well as issue management aspects.

Once the programmer gives the link of object-oriented project using relevant syntax, the java parser (open source solutions for parsing source code) extracts the basic metrics that are needed to find the main code smells. The code smell detection algorithms are based on these basic metrics and after getting those basic metrics, the code smell detection process releases the reports related to code smells where the end user can access.

Software project may store in s local machine or in a remote version control repository like github or bitbucket. The DSL is capable of producing output for both the cases mentioned below.

Local Project – Project, which is located in the local machine where the analysis takes place.

```
MProject p1 type:local
src:"E:\\\\eclipse_ws2\\\\Newwss"
analyze:false
MClass c1=p1/"AddCar"
MPrint p1.longParameterList
MPrint p1.c1.functionDetails
```

Figure 2: Sample Code, Analyzing a local project

Remote Project – Project, which is located in remote server like Github. For example, if the programmer gives a Github project link using the syntax of DSL, then the particular project will be downloaded to the local machine.

```
MProject p1 type:git
src:"https://github.com/
     SquareSquash/java.git"
analyze:true
MPrint p1.duplicationDensity
```

Figure 3: Sample Code, Analyzing a remote project

In the output stage, the programmer or end user type in her query program (using the DSL) with relevant syntax on the given text editor. Based on the query, the language produces reports accordingly. The language is fully dynamic and the output will be obtained based on the query typed in by the programmer. For example, if the programmer wants to get all the code smells in the given object oriented project, she needs to write the query by using the syntax of our DSL. Then all code smells are displayed in the coding interface as a text output in console. Likewise the analyzed reports can be taken as for the whole project or as component wise of a project (for a java project, DSL supports the class level analysis as well).

4. RESULTS

During this research, we have developed a simple domain-specific language and it was presented as an eclipse IDE plugin. Nowadays eclipse is used for the coding in industry as a major IDE with higher rank of user-friendliness. So the solution can be easily plug into the coding interface.

The new language is capable of detecting a set of code smells by writing a minimum number of source lines. As of now, god class, long method, feature envy, lazy method, lazy class and data class can be detected from this language. The results have been compared with existing code smell detection tools and from that we obtained similar results.

The DSL was developed to detect most of the recognized code smells in many aspects and it is capable of analyzing software projects in a manner that taking the URL for project location or repository location as the input. One of the notable advantages of our approach is that by writing small and simple queries, analysis tasks can be manipulated easily.

5. CONCLUSION

Most of the available static analysis tools are standalone applications, which make programmers access these tools separately to perform various software analyses. However, expert programmers often prefer coding interfaces rather than accessing various other tools for their daily analyzes tasks.

Software analyzing techniques as well as code smells detection techniques are ever changing depending on the requirements and demands of software practitioners and researchers. Therefore, the language should be capable of adapting to the changes while keeping a stable blue print. So the change adaptation process should be automated in a way that can easily plug into the infrastructure. For example, integration with Jira like project management tool would be an interesting future direction.

As future work, we hope to introduce a service-oriented API where anyone can contribute for the evolution process of our initial infrastructure. Reporting mechanism needs to be further improved together with a refactoring plug-in to make the programmers life easier.

6. REFERENCES

[1] L. Panagiotis, “Static Code Analysis”, IEEE Software, vol. 23, pp.58-61, 2006.

[2] F. A. Fontana, P. Braione, and M. Zanoni, “*Automatic detection of bad smells in code: An experimental assessment*”, “Journal of Object Technology”, University of Milano-Bicocca, Italy, 2011.

[3] M. Lanza and R. Marinescu. “*Object-Oriented Metrics in Practice*”. Springer-Verlag, 2006. Doi: 10.1007/3-540-39538-5.

[4] Martin Fowler. Refactoring: “*Improving the Design of Existing Code*”. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1999.

[5] F. Margus, "*Domain Specific Languages in a Customs Information System*". IEEE Software. doi:10.1109/MS.2009.152. ,1 January 2009.

[6] R. Dyer, H. Nguyen, H. Rajan and T. Nguyen, "*Boa-Ultra Large Scale Repository and Source Code Mining*", ACM Transactions on Software Engineering and Methodology, vol. 25, no. 1, pp. 1-34, 2015

[7] van den Bos, M. Hills, P. Klint, T. van der Storm and J. Vinju, "*Rascal: From Algebraic Specification to Meta-Programming*", Electronic. Proceedings in Theoretical Computer. Science, vol. 56, pp. 15-32, 2011.

[8] C. Wijesiriwardana, G. Ghezzi, and H. Gall. “*A guided mashup framework for rapid software analysis service composition*”. In Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, volume 1, pages 725–728. IEEE, 2012.

[9] S. E.. Flensburg, “*oAW xText: A framework for textual DSLs*”, Version 1.1, Sept 22, 2006.