ConcernMeBS: Metrics-based Detection of Code Smells

Hayllander Blonski, Juliana Padilha, Marina Barbosa,
Diogo Santana, Eduardo Figueiredo

Computer Science Department, Federal University of Minas Gerais (UFMG), Brazil

{hblonski,juliana.padilha,marinamb,diogo.marques,figueiredo}@dcc.ufmg.br

Abstract. *Software metrics have been traditionally used to evaluate the maintainability of the software systems and to detect code smells. Code smells are symptoms that may indicate something wrong in the system code. Recently, concern-sensitive metrics and metrics-based heuristics have been proposed to detect code smells. However, the application of this kind of metrics and heuristics are time consuming without proper tool support. To address this task, this paper presents a tool, called ConcernMeBS, to help developers to detected code smells. Based on concern to code mapping, ConcernMeBS automatically finds and reports classes and methods that are prone to suffer from code smells in OO source code.*

1. Introduction

The modularization of the driving design concerns is a key factor to achieve maintainable software systems [Kiczales et al 1997; Parnas 1972]. A concern is any important property or area of interest of a system that we want to treat in a modular way [Robillard and Murphy 2007]. However, certain concerns, called crosscutting concerns [Kiczales et al 1997], might end up being scattered and tangled with each other. Business rules, distribution, persistence, and security are examples of typical crosscutting concerns found in many software systems. The inadequate separation of concerns degrades design modularity and may lead to design flaws, such as code smells [Eaddy 2008; Figueiredo et al 2008; Fowler 1999]. Code smells is a mean to diagnose symptoms that may be indicative of something wrong in the system code [Fowler 1999]. Detection of these code smells by programmers is far from trivial and requires effective tool support.

Software metrics are key means for assessing software modularity and maintainability [Chidamber and Kemerer 1994]. The community of software metrics has traditionally explored quantifiable module properties, such as class coupling, cohesion, and interface size, in order to identify code smells in a software project [Chidamber and Kemerer 1994; Lanza and Marinescu 2006; Marinescu 2004]. For instance, Marinescu [Marinescu 2004] relies on traditional metrics to compose strategies aiming to detect code smells. However, some code smells are often a direct result of poor separation of concerns and traditional module-driven measurement cannot detect them.

A growing number of concern metrics have been proposed [Conejero et al 2011; Eaddy 2008; Garcia et al 2005; Greenwood et al 2007] aiming to quantify the key properties of concerns, such as scattering and tangling. Concern metrics quantify properties of concerns realized in one or multiple modules of the source code [Eaddy

2008; Figueiredo et al 2008]. Concern metrics have been applied with different purposes and used in several empirical studies. They are used, for instance, to compare aspect-oriented and object-oriented programming techniques [Conejero et al 2011; Figueiredo et al 2008; Garcia et al 2005] and to identify crosscutting concerns that should be refactored [Eaddy 2008].

In this context, this paper presents a tool, called ConcernMeBS[1], to find code smells in object-oriented (OO) source code. The ConcernMeBS main goal is to support to detect code smells by using a combination of concern metrics [Eaddy 2008] and metrics-based heuristics [Marinescu 2004]. ConcernMeBS extends our previous tool [Figueiredo et al 2009] and was developed based on recent results of our empirical studies [Padilha and Figueiredo 2012; Padilha 2013]. In these studies, we investigate the efficacy of concern metrics and heuristics on the identification of four code smells, namely Divergent Change, Shotgun Surgery, God Class, and God Method. The study results suggest that ConcernMeBS supports the detection of code smells.

The rest of this paper is organized as follow. Section 2 presents the architecture of the ConcernMeBS. Section 3 discusses the design and implementation of this tool. Examples are used to illustrate the tool main functionalities in Section 4. Finally, Section 5 concludes and points out directions for future work.

2. The ConcernMeBS Architecture

ConcernMeBS has about 5 KLOC and aims to support code smell detections. It was developed as an Eclipse plugin and requires some additional plugins in order to detect code smells. Figure 1 presents the architecture of ConcernMeBS and its relationships with three other plugins, namely ConcernMapper [Robillard and Murphy 2007], ConcernMorph [Figueiredo et al 2009], and Metrics Plugin[2]. ConcernMapper allows users to specify a mapping between concerns and syntactic elements in the source code. The second plugin, called ConcernMorph, is used to compute concern metrics. Finally, Metrics Plugin is responsible for collecting the traditional metrics. These three additional plugins are coupled to Eclipse, as ConcernMeBS also is. Our tool also relies on heuristic rules [Marinescu 2004; Padilha 2013] that compose different concern and traditional metrics in order to detect four code smells [Fowler 1999], namely Divergent Change, Shotgun Surgery, God Class e God Method. It is important to observe that the tool can be easily extended - by means of new heuristic rules (see Figure 1) - to support the detection of additional code smells.

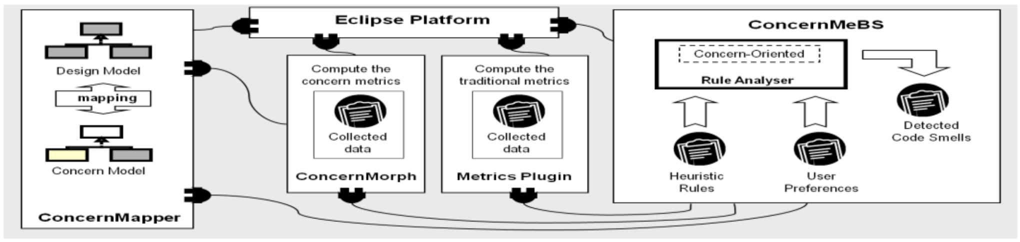


Figure 1. The ConcernMeBS Architecture

[1] ConcernMeBS is available at http://sourceforge.net/projects/concernmebs/

[2] Metrics Plugin is available at http://metrics.sourceforge.net

3. Design and Implementation Decisions

This section discusses some design and implementation decisions we took in the development of ConcernMeBS. Although software metrics constitute a great help to support code smell detection, the isolated interpretation of each metric is not sufficient. That is, if use in isolation, a metric may indicate a code smell but it does not give enough information the code smell, such as its location, type and severity. Therefore, there is a need for the use of heuristic rules. A heuristic rule is a combination of metrics and thresholds to detect specific code smells [Marinescu 2004]. ConcernMeBS relies on heuristic rules to detect concern metrics, as illustrated in Section 3.1. Section 3.2 shows how the tool extends Eclipse.

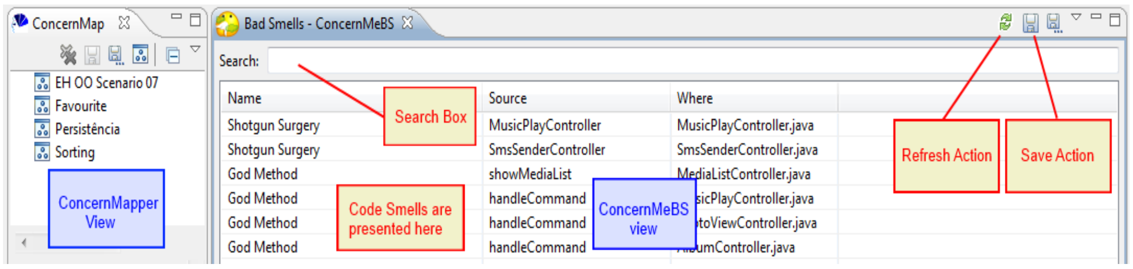
3.1. Eclipse Extensions: Code Smell View and Preference Page

Heuristic 1 below gives a rule example for the detection of God Class [Fowler 1999]. This rule relies on two concern metrics and one traditional metric. The concern metrics used in Heuristic 1 are Number of Concerns per Class (NCC) and Concern Diffusion over Operation (CDO) [Figueiredo et al 2009; Garcia et al 2005]. It also uses the traditional metric Weighted Methods per Class (WMC) [Chidamber and Kemerer 1994]. In short terms, a class is classified as God Class by Heuristic 1 if three conditions are satisfied: (i) it realizes more than two concerns (NCC), (ii) these concerns are scattered in more than thirteen methods, and (iii) this class has more than fifteen methods and parameters.

Heuristic 1 - ***God Class***:
if (NCC > 2) **and** (CDO > 13) **and** (WMC > 15) for a CLASS mapped to concerns
then this CLASS is a GOD CLASS

3.2. Eclipse Extensions: Code Smell View and Preference Page

Figure 2 presents the main view of ConcernMeBS integrated to the Eclipse IDE. This view includes typical buttons, such as *Refresh* and *Update*, which trigger their respective actions (Figure 2). *Refresh* is used to update data being presented in this view while *Save* allows the developer to save the table of code smells in a XML file. Each code smell instance is presented in a row of this view. Columns give additional code smell information, such as its name, a specific source code element in which it is located (source), and the source file (where). If a code smell manifests itself in several elements and files, a detailed list of these elements can been seen by double-clicking in a code smell row.

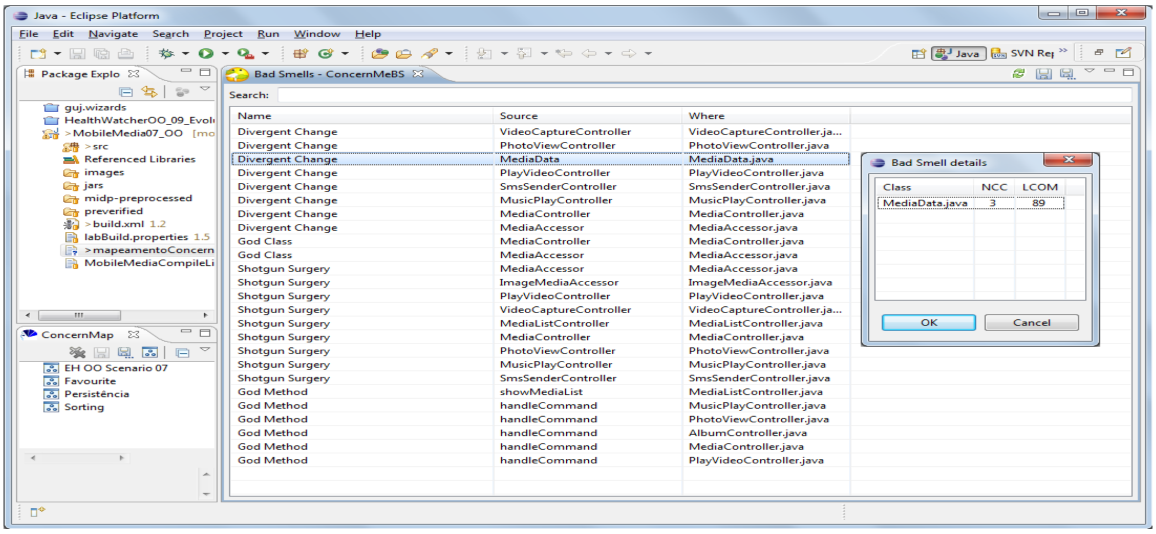


Name	Source	Where
Shotgun Surgery	MusicPlayController	MusicPlayController.java
Shotgun Surgery	SmsSenderController	SmsSenderController.java
God Method	showMediaList	MediaListController.java
God Method	handleCommand	sicPlayController.java
God Method	handleCommand	toViewController.java
God Method	handleCommand	umController.java

Figure 2. ConcernMeBS Main View

In addition to this view, we extend Eclipse with a preference page. Therefore, users can set specific preferences in the ConcernMeBS, for instance, to show or hide information of a specific code smell. The preference page also allows developers to select which code smells and concerns they want to analyze. Another important feature of the ConcernMeBS preference page is the tuning of threshold values for the heuristic rules. This feature gives a flexible way of analyzing systems from different sizes and domains. In fact, a great benefit of ConcernMeBS is that the developer can, in an interactive way, experiment several threshold values and choose those that best fit their project properties. That is, varying threshold values is used to filter the number of code smells candidates.

After performing the concern mapping and tuning threshold values, the tool user can execute the ConcernMeBS plugin to detect code smells. Figure 3 shows a screenshot of ConcernMeBS with the list of code smells detected in the system. Note that the second column in the tool view shows the method or class and the third column presents the hosting file of these elements. By double clicking on a code smell, the values of the metrics used by the heuristic rule to detect that code smell are presented (Figure 3).



Name	Source	Where
Divergent Change	VideoCaptureController	VideoCaptureController.ja...
Divergent Change	PhotoViewController	PhotoViewController.java
Divergent Change	MediaData	MediaData.java
Divergent Change	PlayVideoController	PlayVideoController.java
Divergent Change	SmsSenderController	SmsSenderController.java
Divergent Change	MusicPlayController	MusicPlayController.java
Divergent Change	MediaController	MediaController.java
Divergent Change	MediaAccessor	MediaAccessor.java
God Class	MediaController	MediaController.java
God Class	MediaAccessor	MediaAccessor.java
Shotgun Surgery	MediaAccessor	MediaAccessor.java
Shotgun Surgery	ImageMediaAccessor	ImageMediaAccessor.java
Shotgun Surgery	PlayVideoController	PlayVideoController.java
Shotgun Surgery	VideoCaptureController	VideoCaptureController.ja...
Shotgun Surgery	MediaListController	MediaListController.java
Shotgun Surgery	MediaController	MediaController.java
Shotgun Surgery	PhotoViewController	PhotoViewController.java
Shotgun Surgery	MusicPlayController	MusicPlayController.java
Shotgun Surgery	SmsSenderController	SmsSenderController.java
God Method	showMediaList	MediaListController.java
God Method	handleCommand	MusicPlayController.java
God Method	handleCommand	PhotoViewController.java
God Method	handleCommand	AlbumController.java
God Method	handleCommand	MediaController.java
God Method	handleCommand	PlayVideoController.java

Class	NCC	LCOM
MediaData.java	3	89

Figure 3. ConcernMeBS View in the Eclipse IDE

4. Illustrative Examples

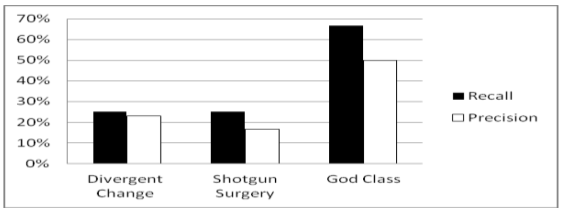
In order to exemplify the ConcernMeBS use, we rely on two systems, named Health Watcher [Greenwood et al 2007] and MobileMedia [Figueiredo et al 2008]. Our results are expressed in terms of Precision and Recall. Precision is calculated by *P = TP/(TP + FP)* and Recall is given by the formula *R = TP/(TP + FN)*. TP means the number of true positives, i.e., code smells correctly identified by the tool; FP means the number of false positives, i.e., incorrectly identified code smells; and FN means the number of false negatives, i.e., code smells incorrectly missed by the tool.

4.1. Results

For the first system, Health Watcher [Greenwood et al 2007], we considered a list of six concerns: (a) *Business:* defines the business elements and rules; (b) *Concurrency:*

provides a control for avoiding inconsistency in the information manipulated by the system; (c) *Distribution:* responsible for externalizing the system services at the server side and supporting their distribution to clients; (d) *Exception Handling:* supports error recovery; (e) *Persistence:* responsible for storing the information manipulated by the system; and (f) *View (GUI):* provides a Web interface for the system.

Figure 4 shows the results of the code smells detected by the ConcernMeBS tool in Health Watcher (left-hand side). The tool has not detected any instance of God Method code smell in this system. In addition, we can see that the results for two code smells, namely Divergent Change and Shotgun Surgery, were below 30% of Precision and Recall. Our best rates for the Health Watcher system regard God Class, in which the tool achieved 50% of Precision and 67% of Recall.



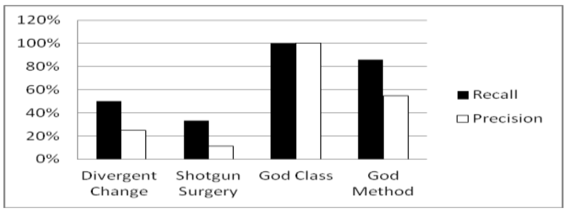


Figure 4. Results for Health Watcher (left) and MobileMedia (right)

For the MobileMedia [Figueiredo et al 2008] system, the five concerns analyzed were: (a) *Sorting:* allows counting the number of times a particular media was viewed by the user and sorting media by the number of views; (b) *Favorites:* allows the user to define a particular media as favorite and to visualize favorite media; (c) *Exception Handling:* implements a mechanism to deal with events that change the normal flow of execution; (d) *Security:* allows passwords to be associated with media albums; and (e) *Persistence:* refers to the ability of the application to retain data between executions.

Figure 4 also presents the results for MobileMedia (right-hand side). As we can see in the figure, the ConcernMeBS tool achieved better results for MobileMedia than it has achieved for Health Watcher. Although there are some Precision rates below 30%, most code smells were detected (i.e., Recall usually above 50%), an exception is Shotgun Surgery with a Recall rate of 35%. On the other hand, both the Precision and Recall for God Class and God Method were higher than 55%. In fact, the tool achieved 100% of Precision and Recall for God Class in the MobileMedia system. This means that all code smell instance were correctly identified.

4.2. Related Tools

Efficiency of software metrics in the detection of code smells has already been a target for many studies. There are many tools available that make use of this concept. Examples of these tools are Together[3] and Codepro[4]. However, metrics utilized in these tools have not been evaluated in experiments with participants. On the other hand, ConcernMeBS was developed based on extensive empirical studies and has the difference of using both traditional and concern metrics. The efficiency of these metrics

[3] Together is available at: http://borland.com/products/Together/

[4] CodePro is available at: http://loose.upt.ro/reengineering/research/codepro

was investigated in our previous experiments. Details of the experiment results are discussed elsewhere [Padilha 2013].

5. Conclusions

This paper presented ConcernMeBS, a tool developed for code smells detection in source codes by the use of concern metrics and heuristics. We summarized the method behind the tool and its main functionalities (Sections 2 and 3) in addition to exemplify how it works (Section 4). Our results so far are, in general, satisfactory, and they reflect the feasibility of using concern metrics and heuristics in the detection of code smells. Nevertheless, our goal for future studies includes the improvement of ConcernMeBS in order to obtain better recall and precision rates for the already supported code smells. We also aim to increase the number of supported code smells.

Acknowledgements

This work was partially supported by Capes/Proex grants 2566/2012, FAPEMIG grants APQ-02376-11 and APQ-02532-12, and CNPq grant 485235/2011-0.

References

Chidamber, S. R. and Kemerer, C. F. (1994) “A Metrics Suite for Object Oriented Design”. Transactions on Software Engineering (TSE).

Conejero, J. M. et al. (2011) “On the Relationship of Concern Metrics and Requirements Maintainability”, Information and Software Technology (IST).

Eaddy, M. et al. (2008) “Do Crosscuting Concerns Cause Defects?”. IEEE Transactions on Software Engineering, 497-515.

Figueiredo, E., et al. (2008) “Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability”. In Proc. of Int. Conference on Software Engineering (ICSE).

Figueiredo, E.; Whittle, J., and Garcia, A. (2009) “ConcernMorph: Metrics-Based Detection of Crosscutting Patterns”. In Proc. Symp. on Foundations of Soft. Eng. (FSE), pp. 299–300.

Fowler, M. (1999) “Refactoring: Improving the Design of Existing Code”. Addison Wesley.

Garcia, A., et al. (2005) “Modularizing Design Patterns with Aspects: A Quantitative Study.” Proc. of Int. Conf. Aspect Oriented Soft. Development (AOSD).

Greenwood, P., et al. (2007) “On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study”. Proc. European Conf. OO Programming (ECOOP), 176–200.

Kiczales, G., et al. (1997) “Aspect-Oriented Programming”. Proc. of European Conference on Object Oriented Programming (ECOOP), 220-242.

Lanza, M. and Marinescu, R. (2006) “Object-Oriented Metrics in Practice”. Springer.

Marinescu, R. (2004) “Detection Strategies: Metrics-Based Rules for Detecting Design Flaws”. In Proc. of Int'l Conf. on Software Maintenance (ICSM), pp. 350-359.

Parnas, D. L. (1972) “On the Criteria to be used in Decomposing Systems into Modules”. Communications of the ACM, 15(12), 1053-1058.

Robillard, M., Murphy, G. (2007) “Representing Concerns in Source Code”, Transactions on Software Engineering and Methodology.

Padilha, J. and Figueiredo, E. (2012) “Detectando Code Smells com Métricas de Interesse”. Workshop de Teses e Dissertações em Engenharia de Software (WTDSoft).

Padilha, J. (2013) “Detecção de Anomalias de Código Usando Métricas de Software”. Dissertação de Mestrado, Universidade Federal de Minas Gerais (UFMG).