2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Poster: Filtering Code Smells Detection Results

Francesca Arcelli Fontana*, Vincenzo Ferme† and Marco Zanoni*
*Department of Informatics, Systems and Communication, University of Milano-Bicocca, Milano, Italy
Email: {arcelli,zanoni}@disco.unimib.it
†Faculty of Informatics, University of Lugano (USI), Lugano, Switzerland
Email: vincenzo.ferme@usi.ch

***Abstract*—Many tools for code smell detection have been developed, providing often different results. This is due to the informal definition of code smells and to the subjective interpretation of them. Usually, aspects related to the domain, size, and design of the system are not taken into account when detecting and analyzing smells. These aspects can be used to filter out the noise and achieve more relevant results. In this paper, we propose different filters that we have identified for five code smells. We provide two kind of filters, Strong and Weak Filters, that can be integrated as part of a detection approach.**

I. Introduction

When we detect design flaws, like code smells, it is important to focus the attention on smells that represent real problems to be removed through refactoring. We can focus the attention on the most relevant results by applying some kind of filters. In this perspective, we define two kinds of code smell Filters of different strength (Strong and Weak) to apply to the detected code smells, with the aim of discarding false positive smell instances, or to provide hints for their possible removal. The proposed Filters can be applied on top of any detection approach, hiding false positives to the users or highlighting properties of reported instances. The aim of Strong Filters is to remove smells to be inspected and refactored, while the idea behind Weak Filters is to highlight a subset of suspected smells, to enable developers or maintainers to undertake, if they want and have time, an evaluation of other possible smells to be excluded. Through Filters, we can identify some cases where smells can be considered domain-dependent or design-dependent smells, and not symptoms of real problems.

We have implemented the Filters for five smells. We evaluated their effectiveness in refining the code smell detection results on 74 systems of the Qualitas Corpus [1].

II. Related Work

Many tools for code smell detection have been proposed, both commercial tools and research prototypes. Few of them provide some kind of filtering facility comparable to the one we define. For example, inFusion[1] provides a kind of filtering mechanism, to allow the user to define some project-specific patterns. The filters have to be manually defined by the user, while in our approach we integrated different kinds of Filters as part of the detection process. In a work of C. Marinescu [2], the author shows how the detection accuracy of two code smells, Data Class and Feature Envy, can be improved by taking into account particularities of enterprise applications. Another work by Ratiu et al. [3] uses the historical information of the suspected flawed structures to increase the accuracy of the automatic problem detection. He considers only God and Data Class smells. In our work, we defined and implemented two kinds of Filter of different categories, for five smells.

III. Code Smell Filters Identification

We figured out the need of defining filters to discard false positive code smells instances while performing experiments on the detection of God Classes and Data Classes on 12 systems belonging to different application domains, taken from the Qualitas Corpus. We used iPlasma[2] to detect the code smells because it is a tool with clearly defined rules and thresholds. We manually checked the results of the detection of the two smells and we realized the importance of taking into account the application domain of the system or other features that are usually not considered during the detection, in order to improve the detection accuracy. In many cases, we classified the results as false positives by considering the following aspects: (a) the used libraries can have an impact on the smell, (b) the smell is a utility class (e.g., Logging) or implements specific functionalities (e.g., Parser), (c) the smell is a test class or (d) Data Classes that are Java beans and God Classes bean managers.

We then extended the analysis to 74 systems belonging to the Qualitas Corpus and to five code smells, as reported in Table I. The proposed Filters are based on the satisfaction of some conditions, capturing characteristics of false positive instances, as for example the (a-d) discussed above. We defined and implemented two kinds of filters: Strong Filter and Weak Filter. A *Strong Filter* is defined as a property of source code entities that can be automatically computed in a precise manner, e.g., we filter out the test classes by identifying if the class implements well known testing frameworks, as for example JUnit. A Strong Filter removes false positive instances from the detection results. A *Weak Filter* is defined as a property of source code entities that can be automatically computed using a heuristic that may led to false positives, e.g., we tag getter and setter methods identified as Shotgun Surgery by applying some heuristics on the signature and the body of the methods. A Weak Filter can identify a code smell that could not be necessarily a problem for the system quality.

[1]https://www.intooitus.com/products/infusion

[2]http://loose.upt.ro/iplasma/index.html

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.256

TABLE I: DEFINED CODE SMELLS FILTERS

Code Smell	Defined Filters	Categories	Type of Filter
God Class	GUI Library	Library Implementer or Extender	Strong
	Test Class	Library Implementer or Extender, Method Caller	Strong
	Entity Modeling Class	Library Implementer or Extender	Strong
	Parser Class	Name Matcher	Weak
	Visitor Class	Name Matcher	Weak
	Persistence Class	Method Caller	Weak
Data Class	Exception Handling Class	Library Implementer or Extender	Strong
	Serializable Class	Library Implementer or Extender	Strong
	Test Class	Library Implementer or Extender, Method Caller	Strong
	Strong God Class Data	Related CS	Strong
	Logger Class	Name Matcher	Weak
Shotgun Surgery	Exception Handling Method	Library Implementer or Extender	Strong
	Test Class Method	Library Implementer or Extender, Method Caller	Strong
	Getter/Setter Method	Name Matcher	Weak
Dispersed Coupling	Test Class Method	Library Implementer or Extender, Method Caller	Strong
Message Chains	Test Class Method	Library Implementer or Extender, Method Caller	Strong

TABLE II: IMPACT OF FILTERS BY CODE SMELL

Code Smell	Strong Filtered			Weak Filtered		
	Flt.	/Tot.	(%)	Flt.	/Tot.	(%)
God Class	28	/ 836	(3.35)	47	/ 808	(5.82)
Data Class	118	/1,723	(6.85)	18	/1,605	(1.12)
Shotgun Surgery	11	/ 508	(2.17)	177	/ 497	(35.61)
Dispersed Coupling	367	/3,555	(10.32)	–	/–	(–)
Message Chains	40	/ 674	(5.93)	–	/–	(–)

Filters are currently classified in the following categories and capture different kinds of information on:

- *Library Implementer or Extender*: the class that the "smelly" entity extends OR the library that the "smelly" class implements.
- *Method Caller*: the methods that the "smelly" entity calls.
- *Related CS*: relations among different smell types.
- *Name Matcher*: the name of the class and/or its methods. Only Weak Filters can belong to this category.

We defined these categories to classify different types of code smell Filters we implemented for each of the five code smells, as reported in Table I.

Table II reports the impact of the Filters on the detection of code smells on the 74 considered systems. For each code smell, we report how many code smell instances have been filtered by Strong and Weak Filters. For Dispersed Coupling and Message Chains we did not identified any Weak Filter. Filters have different impact on different code smells; Strong Filters appear to be more effective on Dispersed Coupling and less on God Class and Shotgun Surgery, while Weak Filters are very effective on Shotgun Surgery. The differences in terms of impact are due to various factors, e.g., the characteristics of the systems, and the number of Filters applicable to each smell. The application of Strong Filters removed 146 classes and 418 methods from the code smell detection results. Weak Filters, instead, matched 65 classes and 177 methods. The percentage of weak-filtered methods is high for Shotgun Surgery, because there are many getter and setter methods.

IV. CONCLUSION AND FUTURE WORK

We define code smell Filters to remove false positives, proposed as part of a code smells detection approach. We performed some initial assessment, showing that filters provide support to developers having to check the source code. Strong Filters removed only entities not affected by code smells, effectively improving the precision of the detection. Weak Filters, instead, signaled either false positives and borderline situations. We observed that Filters have different impact on different code smells. For the future developments we are working on increasing the number of defined Filters to improve the detection results on more code smells and to extend the Filter categories, exploiting other properties of the specific domain or the design, e.g., adding the detection of annotations to improve the filtering of testing classes and methods.

REFERENCES

[1] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *Proc. 17th Asia Pacific Softw. Eng. Conf. (APSEC'10)*. Sydney, Australia: IEEE, Dec. 2010, pp. 336–345.

[2] C. Marinescu, "Identification of design roles for the assessment of design quality in enterprise applications," in *Proc. 14th Intl Conf. Program Comprehension (ICPC'06)*. Athens, Greece: IEEE, Jun. 2006, pp. 169–180.

[3] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proc. 8th European Conf. Softw. Maint. and Reeng. (CSMR'04)*. Tampere, Finland: IEEE, Mar. 2004, pp. 223–232.