Co-Occurring Code Critics

Angela Lozano, Gabriela Arévalo, and Kim Mens

Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium
alozano@soft.vub.ac.be

Universidad Nacional de Quilmes
CONICET - CAETI / UAI
Buenos Aires, Argentina
gabriela.b.arevalo@gmail.com

Université Catholique de Louvain
Place Sainte Barbe 2
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Abstract. Code critics are controversial implementation choices (such as bad smells or code smells) but at a higher level of detail. Code critics are a recommendation facility of the Smalltalk-Pharo IDE. They aim to achieve standard idioms which allow for a better performance or for a better use of object-oriented building mechanisms. Code critics can be identified at the method- or class-level. We are analyzing in several applications which code critics tend to occur in the same source code entity to see to what extent it is possible to identify controversial implementation choices at a higher level of abstraction.

Keywords: Bad smells, code smells, code critics, Smalltalk, Pharo, empirical software engineering, co-occurrence

1 Introduction

In the context of design and coding any application, there is a plethora of implementation recommendations. Some of these recommendations are given at a high level of abstraction (e.g., low coupling and high cohesion) or while others are very specific level (e.g., classes should not have more than 6 methods). Most of the research on recommending the elimination of controversial implementation choices uses a top-down approach. That is, the recommendations given at a high level of abstraction are disassembled into concrete symptoms until a straightforward detection strategy is reached (e.g., Marinescu's design flaws [2], Gueheneuc/Moha's antipatterns [3]). However, it is difficult to argue that those concrete detection strategies and the way in which they are combined represent *all and only those entities* that the high level recommendation aims to convey. We propose to extract high-level recommendations from the analysis of specific recommendations. The recommendations extracted would not be affected by different interpretations (as opposed to the approaches to detect Fowler's bad smells[1] which differ on heuristics[1], metrics[2] and thresholds[3]). Moreover, this study would allows us to validate the need of the specific recommendations analyzed.

[1] Heuristics are incomplete by definition

[2] The definition of some metrics are also open to interpretation resulting in different tools that provide different results for the same metric.

[3] Thresholds tend to be absolute values that cannot be used across applications or relative values whose cut point is arbitrary.

1.1 Code critics

Code critics is a list of implementation choices in Smalltalk known for being 'ungraceful'. These critics may point out at defects or performance issues in the code. There are code critics only for methods or for classes. Each critic has a short name and a description that explains why that implementation choice could be harmful. In some cases, the code critic also proposes a refactoring.

Although code critics may contain many false positives, the IDE allows to 'turn off' manually any result. The results that have been turned off are saved within the image so that the developer does not have to browse the same false positive ever again. Each code critic belongs to a category that indicates its harmfulness. The categories are Unclassified, Style, Idioms, Optimization, Design Flaws, Potential Bugs, and Bugs.

2 Data collected

Although the code critics tool is designed to analyze the whole image on a selection of critics, as anything else in Smalltalk it can be run programmatically. We analyze all critics implemented except Spelling rules [4](i.e., 120 code critics) from which 27 apply to classes, and 93 apply to methods. We analyzed all packages contained in the image used for the latest distribution of Moose (i.e., Pharo 1.4). For each package (71 in total) we find all critics in methods/classes except for those that implement tests[5]. The result of this analysis is a set of boolean tables (two tables per package: one for its methods and another one for its classes) that indicate which source code entities had which critics (each row has a code critic while each column has a method or class of the package) (shown in Table 1).

These boolean tables are converted into distance tables that measured to what extent the entities affected by one code critic are also affected by another code critic. The distance between two code-critics are calculated by counting the number of source code entities (classes or methods) that were affected by only one of them, over the number of source code entities (classes or methods) that any of them affected. For instance, the distance between cc1 and cc2 is 1 (first cell in Table 2) because their results differ for two classes. We see in Table 1 that cc2 affected only class2, while cc1 affected only class3, out of the two classes that were affected by any of these critics: class2 and class3.

Based on the boolean Table (shown in Table 1) and the distance table (shown in Table 2) we proceed to discard pairs of critics that do not seem interesting

[4] Spelling rules check the spelling on the identifiers of classes, methods, variables, and comments. Given that these violations do not refer to the structure or design of the the source code we discard them because they are likely to generate noise in the results, and are non-critical for software development.

[5] Tests were excluded because critics to their code are likely to be false positives. For instance, duplicated code (which may occur due to calls to assert or other testing methods) does not necessarily create hidden links to other test methods. Moreover, test code tends to contain trial-and-error code which does not follow standard coding practices.

	class1	class2	class3
cc1	0	0	1
cc2	0	1	0
cc3	0	1	1
cc4	1	0	0
cc5	1	0	1
cc6	1	1	0
cc7	1	1	1

Table 1. Code critics (cc) per class for a fictitious package.

	cc1	cc2	cc3	cc4	cc5	cc6
cc2	1.0	-	-	-	-	-
cc3	0.5	0.5	-	-	-	-
cc4	1.0	1.0	1.0	-	-	-
cc5	0.5	1.0	0.6	0.5	-	-
cc6	1.0	0.5	0.6	0.5	0.6	-
cc7	0.6	0.6	0.3	0.6	0.3	0.3

Table 2. Distance among code critics shown in Table 1

for our analysis. Three criterion are used to discard pairs of code-critics. First, pairs with high distances (greater than 0.9) are discarded as they do not tend to co-occur and therefore are unlikely to represent a recommendation of a higher level of abstraction. Second, pairs that occur *always* in the same source code entities because they are likely to be different implementations of the same code critic. Third, all pairs for which one of the code-critics covers more than 90% of the source code entities analyzed because they will be automatically correlated with all other code-critics and these relations are likely to generate only noise.

3 Results

We have generated graphs depicting the frequency in which code-critics appear and the strength of their relation. The strength of the relation between a pairs of critics is defined by its frequency (i.e., number of packages where the pair of critics appears) and by the average of their distance (i.e., distance between a pair of code-critics for all packages analyzed). The pairs with lowest distance and highest frequency are analyzed first because they might reveal a undesirable implementation pattern of a higher level of abstraction. So far we have identified four patterns of relations between pairs of code critics. The first pattern occurs when the critics are redundant. This happens when the critics find similar problems. In particular the following pairs refer to code critics in which only the first one provides a refactoring for the critique:

- '*detect:ifNone: − > anySatisfy:*' vs. '*Uses detect:ifNone: instead of contains:*'
- '*Replace with allSatsify:, anySatisfy: or noneSatsify:*' vs. '*Uses do: instead of contains: or detect:'s*'

– *'Rewrite super messages to self messages when both refer to same method'* vs. '*Sends different super message*'

The second pattern occurs when the critics positively contribute to another one, without being an implication (i.e., solving one critic does not solve the other). This happens when the critics have a common root cause. For instance,

– '*Excessive number of variables*' vs. '*Excessive number of methods*'
– '*Sends questionable message*' vs. '*Excessive number of methods*'

The third pattern occurs when the both critics need to be refactored. This happens when the critics have a common root cause. For instance,

– '*Instance variables not read AND written*'[6] vs. '*Variables not referenced*'[7]
– '*Subclass responsibility not defined*'[8] vs. '*References an abstract class*'[9]

The last pattern occurs when the code critics occur often together but they are more useful as a separate rule (as it is more specific). For instance, '*Inconsistent method classification*' vs. '*Unclassified methods*' which would be more useful as '*inconsistently unclassified methods*'.

Acknowledgments. Angela Lozano is financed by the CHaQ project of the Innovatie door Weteschap en Tecnologie.

References

1. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.
2. R. Marinescu. Detecting design flaws via metrics in object oriented systems. In *Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 173–182. 2001.
3. N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *Proc. of the Int'l Conf. on Automated Software Engineering (ASE)*, pages 297–300. IEEE Computer Society, 2006.

[6] This critic should be divided to discriminate between variables readOnly, writeOnly, or notReferenced
[7] This critic hould refer to class variables only (instance variables would be caught by the noReferenced subcritic of the other code critic)
[8] This critic should be refined because as long as all leafs can see an implementation of the method it should not be a bad-smell.
[9] This critic should be refined into 'refers to an abstract class' and 'uses it as instance or with isKindOf':.