Extending a Taxonomy of Bad Code Smells with Metrics

Raúl Marticorena[1], Carlos López[1], and Yania Crespo[2]

[1] University of Burgos
Area of Languages and Computer Systems, Burgos (Spain)
{rmartico, clopezno}@ubu.es
[2] University of Valladolid
Department of Computer Science, Valladolid (Spain)
yania@infor.uva.es

Abstract. Bad Smells define in an informal way code flaws, in order to suggest refactorings, their aim is to improve the design of the code. Current taxonomies group code smells, making use of similarity or correlation criteria between them, and leading to a manual revision of the code. By other side, it is suggested the assistance of using metrics in the detection of bad smells. Metrics can be collected automatically helping to suggest the presence of flaws. Nevertheless, current taxonomies do not link these concepts.
This work tries to establish additional criteria when we want to classify bad smells. These criteria are also related to metric features. Following the current classifications, we propose a method to evaluate the suitability of the tools assisting bad code smell detection, as well as selection and implementation of metrics linked with bad code smells.

Key Words: metrics, refactoring, taxonomy, flaw design, bad smell

1 Introduction

Bad smells are defined as symptoms that point the need of refactoring [4]. Fowler gives his definitions in a descriptive way, suggesting hints that could show the presence of flaws. Bad smells collect design or code flaws , but their detection is based on certain programmer intuition.

An approach to the definition of bad smells is tackled using metrics and heuristics. However, current taxonomies do not settle relations between bad smell groups and used metrics. It seems interesting, at the moment of estimating the usefulness of a collecting metric tool, that we could establish the number of detected smells.

Until now, it is not used additional information in the taxonomies, grouping bad smells depending of certain similarity, more or less subjective. Adding additional features that allow to match bad smells and metrics, could aid to the construction of metric collecting module and selecting them for their implementation.

The remainder of this work is organized as follows: Sec. 2 shows the current works in the bad smell detection and taxonomies, Sec. 3 shows the added criteria to the taxonomy. Sec. 4 establishes some examples applying the extended taxonomy combining it with metrics. We finish in Sec. 5, giving some conclusions of the proposed solution.

2 Related Works

First bad smell definitions were fulfilled in a descriptive way [4], following the style of other "cook books". This presentation enumerates the list of bad smells, without giving a classification or clustering criterion. The author proposes that *"In our experience no set of metrics rivals informed human intuition."*. However, in the presence of a big volume of code, is difficult to manage a manual detection process of bad code smells, based on the intuition. Currently, there is available in the web [3] a simple classification in *"within classes"* and *"between classes"*. This is the first division that we find in [8], although this author subdivides them into a bigger number of groups, since he adds a great number of smells not included in [4]. He focuses on heuristics for their detection.

In [5], Mantyla proposes other taxonomy based on similarity criteria on smells defined in [4], and validated with the survey results of several programmers. From six categories at the beginning, currently it has been reduced to five groups: Bloaters, Object-Oriented Abusers, Change Preventers, Dispensables and Couplers. It is specially interesting that he proposes the use of metrics in the detection of some bad code smells.

Other works, using as basis the bad smell definition in [4], propose the use of queries on logic predicates [2], or queries on a database [6] to detect code flaws. Again in these works, it is settled the use of metrics, although not a guide to their selection, since it is not used any bad smell taxonomy.

Therefore, from previous taxonomies and attempts of using metric for their detection, we want to extend that taxonomy with additional features, making easier selection processes.

3 Extended Taxonomy of Bad Smells

From the previous taxonomy in [5], we keep its groups, although choosing the last version by simplicity. In that taxonomy, it is not taken into account features that will be pointed in the extended taxonomy. It allows us to cross the data from metrics.

3.1 New Features to Apply

Features not explicitly included in the previous taxonomy are:

[3] http://wiki.java.net/bin/view/People/SmellsToRefactorings

Granularity size of the component that suggests bad code smells. In the case of object-oriented languages, we can classify the following levels: system (package, namespace, cluster, etc.), class and method.

Intra vs. Inter-relations (Intra) the bad smell could be observed from the individual observation (intra) of the component, without having available information of other related components.

Inheritance (IH) information about inheritance hierarchy is needed to suggest the bad smells.

Access Modifiers (Acc) access level among the components.

Using these new features, we extend the existing taxonomies (see Table. 1), to obtain a new extended taxonomy where is pointed out the presence or absence of each feature (**Y**es/**N**o). How we can observe, some of the criteria are partially applicable. In the case of the granularity, almost all defined bad code smells are focused in class and method flaws, so the system level is not applicable at this moment.

Table 1. Extended Taxonomy

Bad Smell	**Group**	**Gran.**	**Intra**	**IH**	**Acc**
Data Clumps	Bloater	Class	N	N	N
Large Class	Bloater	Class	Y	N	N
Long Method	Bloaters	Method	Y	N	N
Long Parameter List	Bloaters	Method	Y	N	N
Primitive Obsession	Bloaters	Method	N	N	N
Alternative Classes with Different Interfaces	O-O Abusers	Method	N	N	N
Refused Bequest	O-O Abusers	Class	N	Y	N
Switch Statements	O-O Abusers	Method	Y	N	N
Temporary Field	O-O Abusers	Class	Y	N	N
Divergent Change	Change Preventers	Class	N	N	N
Parallel Inheritance Hierarchies	Change Preventers	Class	N	Y	N
Shotgun Surgery	Change Preventers	Class	N	N	N
Data Class	Dispensables	Class	Y	N	N
Duplicate Code	Dispensables	Method	N	N	N
Lazy Class	Dispensables	Class	Y	N	N
Speculative Generality	Dispensables	Class	Y	Y	N
Feature Envy	Couplers	Method	N	N	Y
Inappropriate Intimacy	Couplers	Class	N	N	Y
Message Chains	Couplers	Class	N	N	N
Middle Man	Couplers	Class	N	N	N
Comments	Not Defined	Class	N	N	N
Incomplete Library	Not Defined	Class	N	N	N

3.2 Metric Features

Software metrics are widely extended, from a theoretic and practical point of view. Most of development tools include metric collection to aid the audit of the system.

In our particular case, we only focus on applicable metrics with code, and specially with object-oriented metrics. We can apply the selected features related to bad smells on these metrics: Granularity, Coupling (Intra. vs. Inter-Classes), Inheritance, Access and Abstractness.

For instance, taking metrics of a tool as RefactorIt [1], that allows to refactor java code using metrics, and following its own classification, we have [4]:

Table 2. Metric Classification

Metric	**Desc.**	**Gran.**	**Intra**	**Inh**	**Acc**
CLOC	Comment Lines of Code	Class	Y	N	N
V(G)	McCabe's Ciclomatic Complexity	Method	Y	N	N
NP	Number of Parameters	Method	Y	N	N
DIT	Depth of Inheritance Tree	Class	N	Y	N
RFC	Response for Class	Class	N	N	N
...	...	...	...	...	...

4 Usability Example of the Extended Taxonomy Mixing Metrics

The problem to be solved is: with current taxonomies and different metric collection tools, ***which tool can help us to suggest the presence of a greater number of bad code smells?***

So far, this selection was based on the human intuition. New added features should help to the user to chose one or other tool. In our case, we compare an Eclipse plugin to obtain metrics (Metrics-1.3.6), a refactoring tool that includes metric calculation (RefactorIt 2.5) and a reengineering software tool as DMS (Semantic Design), that includes the retrieval of a metric set.

The process is as follows: we build a general table as in Table. 2, with metric definitions. For each one of these tools, we complete their own list of metrics linking them with the general definition, solving possible name conflicts. For example: Number of Parameters is NP in RefactorIt and NOP in Eclipse, but both of them should share their features.

These data are crossed (using a SQL join between tables, comparing the defined properties and foreign keys). As final result, we obtain an association

[4] For shake of brevity, we omit all metrics. A larger description is available in http://www.refactorit.com/?id=1376

between bad code smells and metrics. Each bad smell is linked with a metric set of which features could help to detect them.

For example, if we select the “Switch Statements” bad smell, Mantyla [5] proposes to use of NLOC (Number of Lines of Code) and V(G) (Cyclomatic Complexity) to detect it. When our method is applied, obtained metric set is greater than his suggestion, including metrics (taking the three tools) as: Number of Parameters, Nested Block Depth, Halstead Measures, NLOC and V(G). Although there are metrics not useful to suggest the bad smell, other metrics, as Nested Block Depth, could improve its detection.

In not ideal situations, some bad smells will not have metrics that could help to suggest them. In the following table (see Table. 3), we show the results, comparing the number of bad code smells not detected in each group (following the initial taxonomy in [5]).

Table 3. Bad Smell Coverage with Metrics

Group	**BS Number**	**Not covered (Eclipse)**	**Not covered (RefactorIt)**	**Not covered (DMS)**
Bloaters	5	2	0	2
O-O Abusers	4	1	1	2
Change Preventers	3	2	0	3
Dispensables	4	1	1	2
Couplers	4	4	2	4
Not Defined	2	1	1	1

The resultant comparative shows us that metric set in RefactorIt seems the most adequate set to detect smells. It is not surprising, since these metrics are integrated in a refactoring tool (from the first moment, the tool was designed with this aim), while Eclipse plugin does not have this as its main aim, and DMS tool does not focus on refactorings as fundamental aspect of its design, seeing that its functionality is wider.

This process could be improved in several ways. First of all, new features could be added to smells and metrics. The number of bad smells can be increased, and new smells could need these new features. We should mark that current solution is easily extendable to these new properties. By other side, we compare the number of bad smells not covered. More accurate analyses need to detach the metrics associated to each one, as well as their suitability to detect the concrete bad code smell.

5 Conclusions and Future Works

Admitting the usefulness of current taxonomies, we try to contribute with other criteria and methods to select metric collection tools. In this work, we also consider metrics as way to suggest smells that could drive to refactorings. By means of this procedure, we can cover a reengineering process, integrating all these elements. Obviously, the procedure is not finished, it is a first approximation of selecting metrics and relate them to smells. By other side, applying of this method lead us to choose more useful metric sets in the construction of tools, using them as support.

In previous works [3, 7], we showed the usefulness of using metrics in this process, and the need to integrate modules collecting metrics. However, metric selection to be implemented is not closed yet.

The proposed method establishes a decision system, although could be improved. This procedure is not only applicable to metrics, but we can combine it with other related terms, heuristics, in future works.

References

1. Refactorit - java refactoring tool. http://www.refactorit.com, 2006. Web Resource.
2. Francisca Muñoz Bravo. *A Logic Meta-Programming Framework for Supporting the Refactoring Process.* PhD thesis, Vrije Universiteit Brussel, Belgium, 2003.
3. Yania Crespo, Carlos López, Raul Marticorena, and Esperanza Manso. Language independent metrics support towards refactoring inference. In *9th ECOOP Workshop on QAOOSE 05 (Quantitative Approaches in Object-Oriented Software Engineering). Glasgow, UK. ISBN: 2-89522-065-4*, pages 18–29, jul 2005.
4. Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Number 0-201-48567-2. Addison-Wesley, 2000.
5. Mika Mäntylä. *Bad Smells in Software - a Taxonomy and an Empirical Study.* PhD thesis, Helsinki University of Technology, 2003.
6. Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of the TOOLS*, USA 39, Santa Barbara, USA, 2001.
7. Raul Marticorena, Carlos López, and Yania Crespo. Parallel inheritance hierarchy: Detection from a static view of the system. In *6th International Workshop on Object Oriented Reenginering (WOOR), Glasgow, UK.*, page 6, jul 2005. http://smallwiki.unibe.ch/woor/workshopparticipants/.
8. William C. Wake. *Refactoring Workbook.* Addison-Wesley, 2003.