Impact of Refactoring on Quality Code Evaluation

Francesca Arcelli Fontana
Università of Milano Bicocca
Department of Computer Science
Milano, Italy
arcelli@disco.unimib.it

Stefano Spinelli
University of Milano Bicocca
Department of Computer Science
Milano, Italy
spinelli@disco.unimib.it

ABSTRACT
Code smells are characteristics of the software that may indicate a code or design problem that can make software hard to understand, to evolve and maintain. Detecting code smells in the code and consequently applying the right refactoring steps, when necessary, is very important for improving the quality of the code. In this paper, according to well known metrics proposed to evaluate the code and design quality of a system, we analyze the impact of refactoring, applied to remove code smells, on the quality evaluation of the system.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement – *Restructuring, reverse engineering and reengineering*; D.2.8 [**Software Engineering**]: Metrics – *Product metrics.*

General Terms
Measurement, Design.

Keywords
Refactoring, Code smells, Metrics, Quality evaluation.

1. INTRODUCTION
To improve the code and design quality of a system, we can perform different actions; in this paper we focus our attention on the detection and removal of code smells. Removal is done according to the right choice of refactoring ([13, 14. 15]).

Smells have been defined in ([5]), others have been identified in the literature later and new ones can be discovered. Different tools for code smell detection have been developed that exploit different detection techniques. Usually the detection techniques are based on the computation of a particular set of combined metrics ([10]), standard object- oriented metrics or metrics defined ad hoc for the smell detection purpose.

An important issue regards the choice of the metric's threshold values. To establish these values different factors have to be taken into account, such as system domain and size, expertise and personal background knowledge of the software engineers who set these values (for example the threshold for the Long Method or Large Class smell when we have to consider a method long or a class large). Changing these thresholds obviously has a great impact on the number of detected smells (too many smells or lost smells).
Usually other information for code smells detection, other than metrics combination, is not taken into account, as for example relations existing between classes and other smells.
Most of the tools are not able to perform automatic refactoring to remove smells, Jdeodorant [18] is an example of a tool described below which provides refactoring choices, but modern IDEs are usually capable of performing refactoring automatically, also if they need user guidance.

If removing code smell is useful for improving the quality and maintainability of a system, what are the most critical smells? and hence which are the first ones to be removed?
Identifying the most critical smells is not an easy task and certainly depends on several factors. In this paper we describe our approach, where we consider different quality metrics, which we introduce in Section 2, and we evaluate the impact of the refactoring, applied to remove code smells. on the values of these metrics. We consider metrics proposed in the literature to measure cohesion, coupling and complexity of a system. We start from six metrics belonging to these categories, but we can obviously consider other metrics in future experimentations. We decided to exploit tools both for automatic code smell detection and for the application of automatic refactoring.

Our approach is based on the following steps:

1. computing the metrics on a software system
2. detecting the code smell
3. automated refactoring for one smell
4. reapplying step 1
5. analyzing the increment/decrement for each metric value
6. reapplying step 3-5 for each smell
7. identifying the smells, which refactoring better improves the quality of the analyzed system.

Obviously, step 1. is useful for gaining knowledge on the overall quality of the system. It is out of the scope of this paper to discuss these aspects. In our investigation we decided to focus our attention only on some smells defined in [5] and to describe the results we obtained by analyzing an open source object-oriented system of about 400 classes.

Our approach follows the one proposed in [17], where the authors propose a framework in which a catalogue of object-oriented metrics can be used as indicator for the transformations to be

WRT'11, May 22, 2011, Waikiki, Honolulu, HI, USA

applied to improve the quality of a legacy system. They use metrics to detect potential design flaws and to suggest transformations to correct them. Here we use metrics to evaluate the positive or negative impact of refactoring on the quality of a system and identify the priorities of the smells to be removed. In [17] for design flaw they consider architectural, structural and behavioural flaws, here we focus on code smells.

The results of our investigation suggest that following the advice of correcting smells by refactoring does not predictably improve other quality metrics.

The paper is organized in the following Sections: in Section 2 we introduce the metrics that we use and the tools to compute them; in Section 3 we describe the smells on which we focus our analysis, the refactoring proposed in [5] to remove them and we briefly introduce the tools for their detection; in Section 4 we summarize the results of our investigation by outlining the impact of the refactoring applied to remove the smells on the metrics values. We follow the steps 1-7 described above. Finally, in Section 5 we conclude and describe some future developments.

2. METRICS FOR QUALITY EVALUATION

Metrics are useful to check if an object-oriented system satisfies some design principles or contains violations of these principles or other kinds of defects, smells, anti patterns that removed could improve the quality of the system.

Below we briefly introduce the metrics we use and compute in this paper following the catalogue of [17]:

Data Abstraction Coupling (DAC**)** [11]: DAC is the number of ADT's defined in a class.

Lack of Cohesion in Methods (LCOM) **[7]**: Consider a class *C*, its set M of m methods $M_1, \ldots, M_m$, and its set A of a data members $A_1, \ldots, A_a$ accessed by M. Let $\mu(A_k)$ be the number of methods that access data attribute A_k where $1 \le k \le a$. Then, LCOM(C(M,A)) is:

$$\frac{(\frac{1}{a}\sum_{j=1}^{a}\mu A_j)-m}{1-m}.$$

Number Of Methods (NOM) [11]: NOM is the number of local methods defined in a class, which may indicate the operation property of a class.

Response For a Class (RFC) [4]: The Response Set for a Class (RS) is a set of methods that can be potentially executed in response to a message received by an object of that class. Mathematically it can be defined using elements of set theory.

Tight Class Cohesion (TCC) [4]: A class can be represented as a collection of abstract methods *(AM)* where each *AM* corresponds to a visible method in the class. The representation of a class using abstract method is called an abstracted class *(AC)* which is a multi-set and can be formally expressed as: *AC(C) = ||AM(M)| MV(C) ||*
where *V(C)* is the set of all visibile methods in a class *C* and in the ancestors of *C*. Let *NP(C)* be the total number of pairs of abstract methods in *AC(C)*. Let *NDC(C)* to be the number of directed connections in *AC(C)*, then TCC is the relative number of directly connected methods which can be expressed as: *TCC = NDC(C) / NP(C)*.

Weighted Methods per Class (WMC)[4]: Consider a class C_1, with methods $M_1, \ldots, M_m$ and $c_1, \ldots, c_m$ are the static complexity of the methods then WMC is:

$$\sum_{i=1}^{n} c_i$$

The metrics LCOM and TCC belong to the category of Cohesion metrics, DAC and RFC to the Coupling category and WMC and RFC to the Complexity category.

We have computed the metrics WMC, LCOM and RFC with *Understand for Java* tool [19] and the metrics TCC, DAC and NOM with *VizzMaintenance* tool [20].

3. CODE SMELLS

We report below the smells and the associated refactoring on which we focus our analysis. We chose smells that have been defined in the Fowler book [5] for which there are automatic tools to detect them; the refactoring to remove the smells has been described in the same book.

3.1 Code Smells and Refactoring

Feature Envy ([5]) smell is a method that seems more interested in a class other than the one that it is in. The method clearly wants to be elsewhere, so uses *Move Method* to get it there. Sometimes only part of the method suffers from envy so in that case you can use *Extract Method* on the jealous bit and *Move Method* to get it home. There are several design patterns ([6]) that break this rule, including: Strategy, Visitor and Delegation patterns. These patterns are used to combat the divergent change smell.
Long Method: the longer a procedure is, the more difficult it is to understand. Nearly all of the time all you have to do to shorten a method is *Extract Method*. If you try to use *Extract Method* and end up passing many parameters, you can often use *Replace Temp with Query* to eliminate the temps and slim down the long list of parameters with *Introduce Parameter Object* and *Preserve Whole Object* **([5])**.
Shotgun Surgery: this situation occurs each time you make a kind of change, you have to make a lot of little changes to many different classes. When the changes are all over the place they are hard to find, and it is easy to miss an important change. You want to use *Move Method* and *Move Field* to put all the changes in a single class. If no current class looks like a good candidate then create one. Often you can use *Inline Class* to bring a whole bunch of behaviors together **([5])**.
Large Class ([5]): When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind. A class with too much code is also a breeding ground for duplication. In both cases *Extract Class* and *Extract Subclass* will work.

3.2 Code Smells Detection Tools

Many tools for code smell detection have been provided. We briefly describe below the tools that we have used for the analysis described in this paper. Many other tools are available as for example Décor, CheckStyle, iPlasma, CodeVizard, Stench Blossom.
JDeodorant [18] is an eclipse plug that automatically identifies four code smells in Java programs, proposes the refactoring that resolve the identified problems and automatically applies the most effective refactoring.

PMD [16] scans Java source code and looks for potential problems like: possible bugs, such as dead code, empty try/catch/finally/switch statements, unused local variables, parameters and private methods over complicated expressions, duplicate code, copied/pasted code. Moreover, PMD allows the user to set the metrics thresholds.

InFusion [8] supports the analysis, diagnosis and quality improvement of a system at the architectural, as well as at the code level and covers all the necessary phases of the analysis process, from model extraction, including scalable parsing for C, C++ and Java, up to high-level metrics-based analyses, detection of code duplication and intuitive visualizations. InFusion allows to detect more than 20 design flaws and code smells, like Code Duplication, classes that break encapsulation (Data Class, God Class), methods and classes that are heavily coupled or ill-designed class hierarchies and other code smells. InFusion has its roots in iPlasma ([12]).

We concentrated our attention on these three tools because they are able to detect the smells described in Section 3.1 and we have already experimented them several times ([3]).

In this work we describe the analysis we performed on an open object-oriented system of about 400 classes, GanttProject version 1.10.2, a multi-application platform for planning and project management. In Table 1 we report the number of smells found in this system and the tools to detect them.

Table 1. Tools for code smells detection

Tools	Code Smells	N°
Jdeodorant-	Feature Envy	53
	Large Class	58
PMD	Long Method	46
inFusion	ShotgunSurgery	7

Usually code smell detection tools are not able to perform automatic refactoring. Among the above tools only Jdeodorant is able to do this. For the refactoring of the smells not detected through Jdeodorant we exploited *IntellJIDEA 10* [9].

Table 2. Smells and automatic refactoring

Code Smells	Detection Tools	Refactoring	Automatic refactoring
Feature Envy	JDeodorant	Move Method	Jdeodorant
Long Method	PMD	Extract Method	IntelliJ IDEA 10
ShotgunSurgery	inFusion	Move Method	IntelliJ IDEA 10
Large Class	JDeodorant	Extract Class	Jdeodorant

In Table 2 we report the smells, the refactoring that we decided to apply and the tool we have used for the automated refactoring.

4. IMPACT OF REFACTORING STEPS ON QUALITY EVALUATION

To evaluate the impact of refactoring on the quality of the analyzed system we did the following steps:

- we evaluated the metrics described in Section 2 on the GanttProject system using the tools for the metrics computation, by considering the values obtained in all the classes of the system;
- for each smell of Table 1 detected with the corresponding tool, we applied the refactoring of Table 2 always through the appropriate tool for refactoring;
- we computed again all the metrics values after having applied the refactoring;
- we performed the same above steps for all the smells;
- we evaluated the impact of refactoring by analyzing all the values of the metrics before and after the refactoring.

We report a summary of these results in Table 3, where with – we mean a negative impact on the metric value, with + a positive impact and with = no change in the value. Moreover, we have included the percentage number which shows the deviation of each metric value before and after the refactoring applied to the system.

Table 3. Impact of refactoring on the metrics values

REFACTORING	WMC	LCOM	RFC	DAC	NOM	TCC
Move Method (Feature Envy)	- 0,1%	- 0,1%	- 0,3%	- 0,8%	- 0,4%	+ 14,4%
Extract Method	+ 14,3%	- 1,8%	+ 0,1%	+ 6,3%	+ 3,4%	+ 21,3%
Move Method (Shotgun Surgery)	= 0%	- 10,8%	= 0%	- 0,4%	= 0%	+ 12,4%
Extract Class	- 0,5%	- 0,7%	- 0,9%	- 1,5%	- 1,3%	+ 4,7%

From Table 3 we can observe that the only metric, whose value increased is TCC for all the refactoring applied for smell removal, because these refactoring have an impact in particular on the methods of the system.

WMC metric value improves when Extract Method refactoring is applied, and the system become more stable. We can see that WMC and RFC metrics decrease and increase for the same refactoring and code smell removal, since it is well known that a correlation exists between the two metrics: high RFC values correspond to a higher system complexity.

LCOM values decrease in all the refactoring because respect to the initial state of the system, there are more methods that access the same attributes of the refactored class. For example this occurs with the Extract Method refactoring.

By applying Extract Method refactoring we have an increment in all the values of the metrics except for the LCOM metric. Hence, only through this first analysis we can argue the importance of removing the Long Method smell and applying the Extract Method refactoring.

Obviously, if we consider other smells, the impact on the metrics value changes, by removing for example the Data manger smell, not considered in this paper, we could have a positive impact on the DAC metric.

After applying refactoring, we checked again with the code smells detection tools for the presence of the smells in the code; all the numbers of smells in Table 1 became 0. The refactoring has been correctly applied.

5. CONCLUDING REMARKS AND FUTURE DEVELOPMENTS

Through the analysis we described in this paper we tried to evaluate the impact of refactoring on the code and design quality of a system through the computation of a well known set of metrics. We started from the refactoring applied to remove some code smells and we decided to use tools: for automatic detection of smells, for metrics computation and for applying the refactoring.

We have to remark several aspects that have to be taken into consideration:

- The validity of the tools for code smell detection: we tried many tools for code smells detection ([3]), and as usual, they provided different results, and a benchmark platform is not yet available. Hence, we can have other smells in the system that the tools have not recognized, or we can find false positives. In our analysis we manually check the correctness of the detected smells.
- The validity of the automated refactoring tools: we checked if the refactoring transformation to remove a smell behaved correctly, but it is difficult to check if the refactoring has introduced other defects or smells.
- The validity of the metrics computation tools: we supposed the tools provided safe computation, but we have not compared the obtained values with those obtained with other tools. While for some metrics, as NOM, it is easier to manually check the value, for other metrics it is certainly a more complex task.
- The set of metrics used in this investigation has to be extended with other metrics which are useful for evaluating code and design quality.

For future developments we would like to perform the same analysis on a large set of systems, considering other smells as for example the duplicate code, one of the worst smells, and also adopting other code smell detection tools to compare and validate the results of the detection. We think that the investigation described in this paper can be useful for gaining knowledge and providing a kind of guideline on the first smells to be removed in a system in order to improve its code and design quality. Moreover, in order to improve some metrics values, we can immediately observe which smells, once removed, produce the greatest impact.

We are currently working on code smell detection, and we are developing a module to be integrated into our project Marple [1] (Metrics and Architecture Reconstruction PLug-in for Eclipse). Moreover, a module of Marple has been developed for metrics computations ([2]), currently used for software architecture reconstruction and anti pattern detection. We aim to extend this module for the computation of the metrics cited in this paper and other more useful metrics for the investigation here described.

In the future we hope to provide in the same platform functionalities for code smells detection, automated refactoring and metrics computations to be used for different analysis and improvements on the quality of an existing system.

6. REFERENCES

[1] F.Arcelli Fontana, M. Zanoni, A tool for design pattern detection and software architecture reconstruction, *Information Science Journal,* Elsevier. (2011), doi:10.1016/j.Dec.2011.

[2] F.Arcelli, S.Maggioni, A metrics based detection of Micro Patterns, In *Proceedings of the IEEE WETSOM Workshop, co-located event with ICSE 2010*, Cape Town, May 2010.

[3] F.Arcelli et al, An experience report on using code smells detection tools, To appear in the *Proceedings of the RefTest 2011 Workshop,* Berlin, March 2011.

[4] S. R. Chidamber and C. F. Kemerer. A metric suite for object-oriented design. *IEEE Transactions of Software Engineering*, 25(5):476–493, June 1994.

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

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[7] B. Henderson-Sellers. *Object-Oriented Metrics : Measures of Complexity*. Prentice Hall, 1996

[8] InFusion: http://www.intooitus.com/inFusion.html

[9] IntellJ Idea 10: http://www.jetbrains.com/idea/features/index.html

[10] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[11] W. Li and S. Henry. Object-oriented metrics that pre- dict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[12] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005), Tools Section*, 2005.

[13] T. Mens, T. Tourwé, A Survey of Software Refactoring, *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, February 2004.

[14] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5), 2008.

[15] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Blac How we refactor, and how we know it. In ICSE '09 *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[16] PMD: http://pmd.sourceforge.net/

[17] Ladan Tahvildari, Kostas Kontogiannis, A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations, In *Proc.IEEE Conference of Software Maintenance and Reengineering* (CSMR'03), 2003, Italy.

[18] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *Proceedings of CSMR* 2008, pp 329–331. IEEE Computing Society, 2008.

[19] Understand for Java: http://www.scitools.com/