Assessing the impact of bad smells using historical information

Angela Lozano
The Open University
Walton Hall
Milton Keynes, UK
A.Lozano-Rodriguez@open.ac.uk

Michel Wermelinger
The Open University
Walton Hall
Milton Keynes, UK
M.A.Wermelinger@open.ac.uk

Bashar Nuseibeh
The Open University
Walton Hall
Milton Keynes, UK
B.Nuseibeh@open.ac.uk

ABSTRACT
Our aim is to gain a better understanding of the relationship between bad smells and design principle violations, in order to better identify the root causes of a given set of bad smells and target refactoring efforts more effectively. In particular, knowing which bad smells point to important design problems would help to focus developers' efforts. In this position paper we argue that such knowledge requires the empirical study of the evolution of software systems: on the one hand because design problems and their symptoms take time to develop, on the other hand because we need to relate maintenance activity to bad smells to measure their relative importance. We illustrate how existing studies of the evolution of a particular kind of bad smell, code clones, have led to further insights into the harmfulness of cloning.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Distribution, Maintenance, and Enhancement – *restructuring, reverse engineering, and reengineering*

General Terms
Measurement, Experimentation.

Keywords
Software evolution, empirical study, software maintenance, source cede, software quality, bad smells, design flaws, design defects.

1. QUALITY AND STRUCTURE
In Software Engineering, quality assessment is based on the assumption that a good design has impact on quality attributes, such as maintainability. Traditionally, based on this assumption software quality has been evaluated through structural and process metrics. Although the relationship between software quality and source code quality is not yet well understood [12], there is evidence to support such an assumption. For example, product metrics have been found to reflect external aspects of software quality, such as software defects [3, 27].

In fact, according to Baldwin and Clark [2], the value of an artifact is determined by its functionality, and given that the structure determines and constrains the ease of implementing the artifact's functionality, the value of an artifact depends strongly on its structure. Several authors have proposed *guidelines* or implementation advice to achieve a better structure in terms of the traditional principles of software design; e.g., high cohesion, low coupling, low complexity, etc. For instance, Parnas [22] claims that hiding details insulating the change reduces its negative effects. Meyer [18] proposes to minimize the size and number of interfaces, and to only interact through them. Martin [17] suggests that modules should depend on abstractions, and Gamma et al. [7] identify typical good solutions for typical design scenarios.

Other authors aim to identify typical causes for failures e.g. anti-patterns [4] , and typical opportunities for improving the design structure e.g. bad smells [5]. Examples of anti-patterns include blob classes, poltergeists, lava flows, spaghetti code, cut & paste programming, functional decomposition, etc. Bad smells include cloned code, long parameter lists, large classes, feature envy, etc. For some authors [16, 19], these bad practices are indicators of violations of design guidelines; therefore they call them design defects or design flaws [16]. However, bad smells were proposed as possible indicators of refactorings. Fowler and Beck [5] remark the subjective nature of bad smells and that they cannot be considered as direct causes of software defects. In fact, there is not yet a conceptual framework to explain the difference between design principles and guidelines, or the relationships among design guidelines, or the relationships between design defects and design guidelines, or the impact of those design defects.

We propose to examine the relationship between maintainability and source code quality by evaluating to what extent bad smells affect the ease to maintain, adapt or extend code. Given that between two artifacts that offer the same functionality (i.e. the same value) users and designers prefer the artifact with less costly structure [2], it is important to identify which structural issues affect software cost, which is in part related to maintainability. Besides, design principles are supposed to help on distributing the responsibilities and on isolating the dependencies among entities. Therefore, enforcing their use

IWPSE'07, September 3-4, 2007, Dubrovnik, Croatia Copyright 2007 ACM ISBN 978-1-59593-722-3/07/09...$5.00.

should reduce the application's complexity and, by consequence, the maintenance effort.

2. COMPLIANCE WITH GOOD PRACTICE

In this section we discuss how the automatic detection of bad smells brought new research questions.

Designing is a process of continual evaluation of trade-offs between the flexibility that a structure offers and the functionality required [2]. However, the functionality that a structure can support is restricted to the predictions made when it was defined [23]. Therefore, given that long-lived applications require constant change [13], their designs eventually need to change as well [23]. According to Lehman's software evolution laws [13], anti-regressive work is required in order to retard this aging process. Naïve approaches attempt to solve this problem by trying to define the most flexible architecture possible. However, it is impossible to optimize an architecture for every possible kind of change [23]. Given that correcting all design problems may impossible due to conflicts with each other solution, it is necessary to detect which design problems are the worse so that the refactoring impact is maximized. This issue was first pointed out by Trifu et al [26].

Several authors have proposed models to provide insights about the causes of structural problems. These models cluster smells (symptoms) into design principles (causes), thereby showing to the programmer which ones to remove and how, because the models indicate which design guidelines were disregarded and which smells would probably disappear once the underlying problem is solved.

2.1 Design flaws as maintainability issues

In his dissertation [16], Marinescu presents a method to detect design flaws (bad smells and anti-patterns), but also defines a mapping between several design flaws and software quality attributes. Each design flaw is weighted to represent its impact on the quality attributes it affects. The dissertation establishes the relation of seventeen design flaws to the ISO maintainability factors. Marinescu calls this relation the factor strategy model. For instance, testability is reduced whenever there are any of the following design flaws: god methods, data classes, god classes, shotgun surgery or lack of the singleton pattern.

2.2 Design flaws as symptoms of design illnesses

A similar approach is presented by Trifu and Marinescu [25], using a medical metaphor to reason about source code quality. While code that complies with design guidelines is considered healthy, source code illnesses are types of design guideline violations. There are three types of illnesses related with three areas of the object oriented paradigm: definition of concepts, distribution of intelligence among the concepts and inheritance hierarchies. The symptoms of a source code illness are different design defects (for instance, bad smells). Some design defects are mandatory in order to consider that a source code entity has violated a design guideline. For instance, one illness is abusive conceptualization and occurs when the 'one class=one abstraction' guideline is broken. However, to be considered as a breaker of the guideline, the class must be identified as a god class. The authors present the violation of three design heuristics as composition of ten types of symptoms.

2.3 Design flaws as characteristics of anti-patterns

Moha and Guéhéneuc [20, 21] propose a framework oriented to the identification of anti-patterns. Each anti-pattern is decomposed at several layers of abstraction into design defects (like bad smells), and each design defect is then divided into the properties that characterize it. For instance, spaghetti code is a structural intra-class anti-pattern characterized by the following design defects: procedural names, long methods, methods without parameters, global variables, no inheritance and no polymorphism. The authors have presented the decomposition of four anti-patterns into sixteen design defects, and the design defects were characterized with values of fourteen properties. They plan to use this model to detect and correct anti-patterns with refactorings.

2.4 Reflection on the models of design flaws

Models like the ones described are welcome, as they help to obtain a conceptual framework that relates design principles to design heuristics and design heuristics to design flaws. However, they have three main issues: they are incomplete, subjective and the concepts they provide may be insufficient to characterize real-world problems. Firstly, the approaches mentioned just cover a small fraction of the design flaws and guidelines in the literature. Secondly, the rationale to compose several design flaws is debatable: different researchers have derived different models from similar sets of design flaws. Finally, although concepts like anti-patterns offer a higher level of abstraction, they may impose restrictions on what can be modeled, neglecting alternative ways in which real code problems are presented.

We think that the approaches presented above are valuable contributions to the field, but that one could gain considerable benefits from the analysis of evolution to complete and validate the proposed models.

Nevertheless, evolution has not been completely disregarded in the area. There are some authors that have taken advantage of historical information to identify bad smells while others have used it to evaluate their detection approach. Gall et al. [6] identify logical dependencies among modules that are usually changed together (change coupling). Besides, they find outliers in growth and amount of change, and in the relation between internal and external change coupling. This analysis allows for identification of bad smells like god-classes. Girba et al. [8] use historical information in order to detect bad smells naturally defined in the evolution context, for instance shotgun surgery (when single changes affect indiscriminately several entities of the application) or parallel inheritance (when two inheritance trees are usually changed in the same way). Their approach is based on monitoring how metrics change at different granularity levels along several versions. Ratiu et al. [24] use historical information to improve the accuracy of detection strategies by ratifying only the design flaws present in at least two versions of the application.

We believe that analysis of the evolution of bad smells can go further than detection of bad smells or validation of the ones detected. Given that design problems appear and evolve over time, it is probable that a single bad smell cannot be considered a threat by itself, but it may *degenerate into* a more harmful bad smell or it may *promote the appearance* of more bad smells. It would be useful to know *when* a bad smell becomes harmful, as well as which bad smells *may become* harmful. Furthermore, if any of these kinds of degrading evolution is found it could

contribute to models that aggregate bad smells into design problems, and to characterize the design problems themselves.

3. THE IMPACT OF DESIGN FLAWS

In this section we discuss why evolution adds value as evaluator of the harmfulness of bad smells, but also as a way to find patterns of degeneration and of co-existence of bad smells.

We propose to extract sets of bad smells that frequently appear in close locations and try to relate the bad smells on each set using diverse sources of information in the source code, such as dependency relations or common vocabulary. This information could be analyzed using techniques like clustering or concept analysis in order to see if they predict the frequent sets. This sort of analysis would tackle incompleteness and subjectivity issues in the area by helping on achieving consensus in the composition of bad smells into higher level concepts. Once the consensus is achieved these groups could be confidently related to design guideline violation and could improve the detection of bad smells.

As mentioned before, not even the authors that propose bad smells [5] consider them as an accurate quality indicator. They only recommend human intuition to assess the harmfulness of a bad smell. It may be possible that the harmfulness of design flaws follows some sort of Pareto rule, making it very important to characterize which of them indeed affect the application's quality. Given that most of the automatic detection techniques aim to recommend actions to developers, it is preferable not to recommend useless actions that will make maintenance even more costly. In order to recognize which bad smells can be harmful we propose to track their changes over time to see if, as they evolve, they indeed contribute to the degradation of the application. Such analysis would pursue three main questions.

(i) Can single bad smells be harmful? If so, which ones?

(ii) From when on should a bad smell or a set of bad smells be considered harmful?

(iii) Do the sets of bad smells reflect a common underlying problem?

In order to tackle the first question, we plan to use change and bug information as indicators of the effort required to maintain a code entity. So far we have proposed 5 measurements to evaluate if the evolution of code entities with bad smells is more difficult than for those that do not have bad smells. These measurements are: number and density [15], impact, likelihood and similarity of changes [14]. We expect to apply similar measurements for bug information. Bug information can help to check if code entities with bad smells present more bugs or more severe bugs (e.g. only blocker and critical bugs). Bug and change measurements reinforce and complement each other to assess the harmfulness of bad smells. We would also like to identify the differences between harmful and harmless bad smells by describing them and relating them to other information. For instance, checking if harmful bad smells are linked only to certain developers or to certain modules may help to explain the results obtained with the measurements. In fact, we are currently studying the relation between cloning patterns [10] and changes in some medium open source projects. We are also trying to identify which other bad smells are located in the same code entities as clones or in the entities related by call dependencies.

To tackle the second question we would like to locate variations in the metrics used to locate the bad smells and how they change over time. The variations can be detected using existing approaches [8]. Another alternative is identifying the values for the metrics that detect bad smells whenever bugs can be related to them, and see if there is statistical correlation. For evaluating the groups of bad smells we will look at their frequency and other change and bug measurements.

And to tackle the third question we would like to look for cases in which sets of related bad smells are eliminated within a small time interval and for bursts of restructuring changes related with bug elimination. The groups of bad smells that disappear when a bug is fixed could be good candidates for symptoms of a design problem. Moreover, if this analysis reveals several sequences of refactorings useful for solving the same design problem, a later analysis could help to identify the more frequent restructuring solutions, which ones affect more bad smells, which ones have a longer impact on the application, and if the order of the refactorings affects the effectiveness of the restructurings.

The analysis of the harmfulness of individual and groups of bad smells could help to prioritize their removal taking into account their possible evolution path. Besides, these results could help to achieve a better quality assessment model based on solid grounds that explains the relation of several bad smells with design guidelines. Such model, added to empirical evaluation of bad smells, could help to identify the more important design guidelines and if there should be priorities in the guidelines enforcement. Without this knowledge, the models proposed would be either incomplete or based on harmless indicators. Given that source code repositories permit automatic analysis of the evolution of an application, and that history has been recognized to be a good maintenance predictor [9], analyzing the evolution of bad smells seems an appropriate approach to understand their harmfulness and give insights about their relationships.

As an illustration of our position, we now summarize how the analysis of evolution provided added value to the debate about the harmfulness of a particular bad smell: duplicated code.

3.1 Cloning evolution: a successful example of the added value of evolution analysis

Clones are two or more identical or nearly identical fragments of code. For a long time cloning has been considered harmful because it might indicate lack of abstraction, it increases the code size and adds hidden dependencies among the entities with cloned fragments. However, recent empirical studies based on evolution have found that the rationale for considering cloning harmful is not always applicable.

Kim et al. [11] analyzed to what extent cloned fragments changed in the same way. They found that most of the clones are used as templates to implement new behavior in the system – these volatile clones are described as beneficial. They also found that less than 40% of related clones were changed simultaneously. Given that these findings partially deny common knowledge about cloning impact, they have motivated a new set of analysis on source code and maintenance. A similar study [1] found that clones that evolve in the same way change in a short time range, and that in many cases when the clones do not change in the same way there is an error due to an inconsistent change. We [15] have found that when methods are cloned they tend to change more. However, the number of co-changes in related clones is not higher when such methods share a clone. This counterintuitive result could mean that the increase of changes is not related with the

fact of being cloned. We also analyzed the effect of cloning in the impact, breadth and correlation of changes in four applications [14]. However we have not found any common pattern to sustain the harmfulness of clones.

There is still plenty of work to establish if cloning is harmful. Based on current results, it seems that there are cases in which cloning can be an appropriate design decision [10].

We intend to apply the analysis explained above to several types of bad smells starting with those related to cloning, to evaluate their harmfulness and to check if their relations evidence underlying design problems.

4. CONCLUSION

Several coding practices have been avoided due to their lack of compliance with design principles. However there is little evidence for their assumed negative impact. This paper has argued for analyzing the evolution of these so called design flaws. We have suggested some ways by which this could be done.

Analyzing all design flaws that have been proposed in the literature may reveal relations among bad smells not considered previously. Furthermore, it allows for the validation of the sets of bad smells as symptoms of the same design problem, tackling incompleteness and subjectivity issues. Using historical data, one can group bad smells according to their evolution, analyze evolution measurements for single bad smells and for groups of bad smells, relate flaws with later bugs and evaluate the benefits of suitable restructurings to eliminate them.

Moreover the analysis of the evolution of bad smells could bring new evidence about the relation between source code design and quality. Therefore it could contribute to produce a cohesive framework of design guidelines and typical violations. If such a model is achieved we could assess which guidelines are more important and give priority to the treatment of the corresponding design problem.

5. REFERENCES

[1] Aversano, L., Cerulo, L., Penta, M.D.: How Clones are Maintained: An Empirical Study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*: IEEE Computer Society, 2007

[2] Baldwin, C.Y., Clark, K.B.: Design Rules. vol. 1: The Power of Modularity: The MIT Press, 2000.

[3] Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10) (1996), 751-761.

[4] Brown, W.J., Malveau, R.C., Hays W. McCormick, I., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley \& Sons, Inc., 1998.

[5] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

[6] Gall, H., Jazayeri, M., Krajewski, J.: CVS release history data for detecting logical couplings. In *Proc. of the Int'l Workshop on Principles of Software Evolution (IWPSE)*, 2003, pp. 13-23.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional, 1995.

[8] Girba, T., Ducasse, S., Marinescu, R., Ratiu, D.: Identifying Entities That Change Together. Presented at *Workshop on Empirical Studies of Software Maintenance (WESS04)*, 2004.

[9] Hassan, A.E., Holt, R.C.: Predicting change propagation in software systems. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, 2004, pp. 284-293.

[10] Kapser, C., Godfrey, M.W.: 'Cloning considered harmful' considered harmful. In *Proc. of the Working Conf. on Reverse Engineering*, Benevento, Italy, 2006.

[11] Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In *Proc. of the European Software Engineering Conference*, Lisbon, Portugal: ACM Press, 2005, pp. 187-196.

[12] Kitchenham, B., Pfleeger, S.L.: Software quality: the elusive target. *Software, IEEE*, 13(1) (1996), 12-21.

[13] Lehman, M.M., Belady, L.A.: Program Evolution: Processes of Software Change. London: Academic Press, 1985.

[14] Lozano, A., Wermelinger, M., Nuseibeh, B.: A Revision of the Evil Clone: Measurements to Evaluate the Impact of Cloning in Maintainability. In *submitted to Int'l Conf. of Softw. Maintenance*: IEEE Computer Society, 2007.

[15] Lozano, A., Wermelinger, M., Nuseibeh, B.: Evaluating the harmfulness of cloning: a change based experiment. In *Proc. of the 4rd Int'l Workshop on Mining Software Repositories*: IEEE Computer Society, 2007

[16] Marinescu, R.: Measurement and Quality in Object-Oriented Design. Politehnica University of Timisoara, 2002.

[17] Martin., R.C.: The Dependency Inversion Principle. In *The C++ Report*, 1996.

[18] Meyer, B.: Applying `design by contract'. *Computer*, 25(10) (1992), 40-51.

[19] Moha, N., Bouden, S., Guéhéneuc, Y.-G.: Correction of High-Level Design Defects with Refactorings. In *In Proc. of the ECOOP Workshop on Object-Oriented Reengineering WOOR*, 2006.

[20] Moha, N., Guéhéneuc, Y.-G., Leduc, P.: Automatic Generation of Detection Algorithms for Design Defects. Presented at *ASE*, 2006.

[21] Moha, N., Huynh, D.-L., Guéhéneuc, Y.-G.: Une taxonomie et un métamodèle pour la détection des défauts de conception. In *Actes du colloque Langages et Modèles à Objets*, 2006, pp. 201-216.

[22] Parnas, D.L.: Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2) (March 1979. 1979), 128-138.

[23] Parnas, D.L.: Software aging. In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 279-287.

[24] Ratiu, D., Ducasse, S., Girba, T., Marinescu, R.: Using History Information to Improve Design Flaws Detection. In *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR)*: IEEE Computer Society, 2004, pp. 223-232.

[25] Trifu, A., Marinescu, R.: Diagnosing Design Problems in Object Oriented Systems. In *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, 2005, pp. 155-164.

[26] Trifu, A., Seng, O., Genssler, T.: Automated Design Flaw Correction in Object-Oriented Systems. In *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR)*: IEEE Computer Society, 2004, pp. 174-183.

[27] Yu, P., Systä, T., Müller, H.A.: Predicting Fault-Proneness using OO Metrics: An Industrial Case Study. In *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR)*: IEEE Computer Society, 2002, pp. 99-107.