Visual Detection of Design Anomalies

Karim Dhambri Houari Sahraoui * Pierre Poulin
Dept. I.R.O., Université de Montréal
{ dhambrik | sahraouh | poulin } @iro.umontreal.ca

Abstract

Design anomalies, introduced during software evolution, are frequent causes of low maintainability and low flexibility to future changes. Because of the required knowledge, an important subset of design anomalies is difficult to detect automatically, and therefore, the code of anomaly candidates must be inspected manually to validate them. However, this task is time- and resource-consuming. We propose a visualization-based approach to detect design anomalies for cases where the detection effort already includes the validation of candidates. We introduce a general detection strategy that we apply to three types of design anomaly. These strategies are illustrated on concrete examples. Finally we evaluate our approach through a case study. It shows that performance variability against manual detection is reduced and that our semi-automatic detection has good recall for some anomaly types.

Keywords: Software quality, Software metrics, Visualization.

1. Introduction

Design anomalies introduced in the development and maintenance processes can compromise the maintainability and evolvability of software. As stated by Fenton and Pfleeger [5], design anomalies are unlikely to cause failures directly, but may do it indirectly. Detecting and correcting these anomalies is a concrete contribution to software quality improvement. However, detecting anomalies is far from trivial [10]. Manual detection is time- and resource-consuming, while automatic detection yields too many false positives, due to the nature of the involved knowledge [12]. Furthermore, there are additional difficulties inherent to anomaly detection, such as context-dependence, size of the search space, ambiguous definitions, and the well-known problem of metric threshold value definition.

In this paper, we propose a semi-automatic approach to detect design anomalies using software visualization. We model design anomalies as scenarios where classes play primary and secondary roles. This model helps reducing the search space for visual detection. Our approach is complementary to automatic detection as defined in [10, 12]. Indeed, we specifically target anomalies that are difficult to detect automatically. We use detection strategies that combine automated actions with user-oriented actions. The judgement of the analyst is used when automatic decisions are difficult to make. Such decisions are related to the application context, low-level architecture choices, variability in anomaly occurrences, and metric threshold values.

2. Anomaly Detection Issues

Although they are not defects, *i.e.*, violations of the functional specification, design anomalies can have a tremendous negative impact on many quality characteristics during software evolution [1]. Detecting design anomalies in source code is subject to inherent difficulties that need to be considered when developing a detection approach. In this section, we first present some examples of design anomalies, and then discuss related detection issues.

2.1. Examples of Anomalies

Our use of the term "design anomaly" encompasses likewise violations of design heuristics defined in [13, 11], antipatterns described in [1], code smells documented in [6], and any other situation in a design which goes against known quality heuristics. To illustrate our approach, we briefly introduce three examples of design anomalies.

A ***Blob*** antipattern is found in designs where one large class monopolizes the behavior and other classes primarily encapsulate data. It is characterized by a complex and non-cohesive controller class associated to simple data classes [1].

A ***Functional Decomposition*** antipattern occurs when a class is designed with the intent of performing a single function. The class therefore has one large method that is responsible for all of the implementation of the function. Most of the class attributes are private, and used mainly inside the class. Such classes often have names that denote a

978-1-4244-2157-2/08/$25.00 © 2008 IEEE

function (*e.g.*, *CalculateInterest* or *DisplayTable*) [1].
A ***Divergent Change*** code smell occurs when one class is commonly changed in different ways for different reasons [6].

2.2. Detection Issues

Although there is a consensus that it is necessary to detect design anomalies, our experience with industrial projects showed that there are many open issues that need to be addressed. In the following, we discuss some of the open questions related to anomaly detection.

How to decide if an anomaly candidate is an actual anomaly? Unlike software bugs, there is no consensus on how to decide if a particular design violates a quality heuristic. There is a difference between detecting symptoms and asserting that the detected situation is an actual anomaly. Human intervention is mandatory to understand the detected situation and to decide if a correction is needed.

Are long lists of anomaly candidates really useful? Detecting dozens of anomaly candidates is not always helpful. In addition to the presence of false positives that may create a rejection reaction from development teams, the process of using the detected lists, understanding the anomaly candidates, selecting the true positives, and correcting them is long, expensive, and not always profitable.

What are the boundaries? There is a general agreement on extreme manifestations of design anomalies. For example, consider a program where one class implements all the behavior and a hundred other classes store data. There is no doubt that we are in presence of a *Blob*. Unfortunately, in real life systems, we can find many large classes, each one using some data classes and some regular classes. Deciding which ones are *Blob* candidates depends heavily on the interpretation of each analyst.

Other open questions includes "How to define thresholds when dealing with quantitative information?", "How to deal with situations when an apparent anomaly is motivated by a particular context?", and "How to recover application-domain information needed for detection?".

Considering these questions, our position is that for a subset of design anomalies, the detection is more effective and useful if it is seen as a code inspection task performed by analysts supported by adequate tools. For this subset of anomalies, we propose a visualization-based approach and a tool that can be used by software maintainers to detect design anomalies. On the one hand, the participation of the analyst during detection brings acceptable answers to the previous questions: handling variability, defining boundaries, and making decisions when the context, the semantic, and the threshold values are necessary. On the other hand, the visualization environment allows the analysis of large sets of data in a reasonable time and in an efficient manner.

3. Representation of Detection Information

Our anomaly detection approach is based on features provided by our software visualization framework VERSO [9]. VERSO generates effective 3D representations of large scale OO programs. Software elements are represented by 3D graphical elements distributed on a 2D plane according to the low-level architecture (*i.e.*, package hierarchy). The representations use quantitative (metrics) and structural (relationships) data obtained by in-house reverse engineering tool PADL [7] and metric extraction tool POM [8]. Our detection strategies use four types of information: quantitative, relational, architectural, and semantic. The remainder of this section discusses how we designed our visualization framework to support manual analysis by efficiently providing these types of information.

Quantitative Information. Many design anomalies are defined/detected using quantitative information (*e.g.*, *"large class"* for *Blob* and *"large method"* for *Functional Decomposition*). Quantitative information is captured using metrics extracted from the code. For a particular class, its metric values are mapped to graphical attributes to be observed by analysts. Classes are displayed as 3D boxes, and metrics are associated with three attributes of the 3D box: height, color, and twist (Figure 1 (left)).

After measuring and mapping metrics to graphical attributes, an analyst can visually evaluate if a detection condition applies to a particular class (*e.g.*, if a class is large). This evaluation can be done in two ways. First, as the analyst looks at all the boxes (*i.e.*, classes of the analyzed program), he can consider the global context to decide if a class has a large metric value compared to the others. The alternative is to use a distribution filter with the box plot technique [5]. Such a filter colors classes depending on their position in the distribution according to one metric. Classes having an extremely high value (outliers) appear red. Figure 1 (right) shows the application of the distribution filter on the coupling metric CBO on classes of the *Xerces* class library (http://xerces.apache.org/).

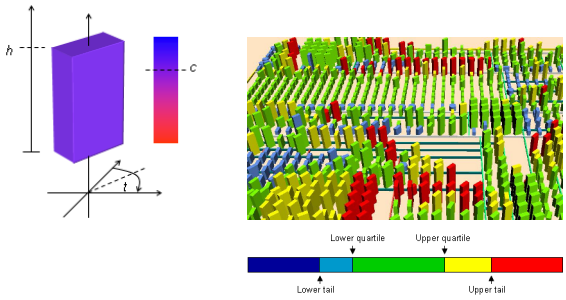

Figure 1. (left) Class Representation. (right) Distribution filter (CBO) on *Xerces*.

Low-level Architecture Information. The definitions of some anomalies include elements that refer to low-level architecture information, *i.e.*, program structure in modules/packages (*e.g.*, a class that interacts mainly with classes from a different package is considered as a *Misplaced Class* anomaly in [10]). As we are interested in Java programs with a tree-like package structure, we visually organize classes using a discrete version of the *Treemap* layout algorithm described in [9]. Figure 2 shows an example of program visualization. When looking at a program representation, we can determine to which package belongs a class.

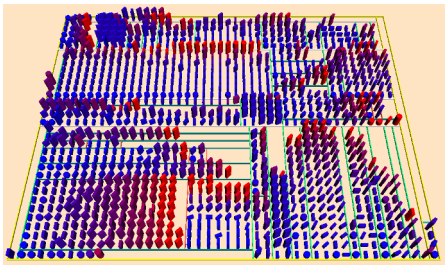

Figure 2. *Xalan* **(1194 classes) in VERSO.**

Relational Information. As mentioned in Section 2.2, design anomalies can be seen as scenarios played by interrelated classes. Different types of relationships are often used in these scenarios (*e.g.*, the definition of the *Blob* considers association and invocation relationships).

In our visualization framework, we use reverse-engineered relationships (associations 'in', 'out', and 'in/out', aggregation, generalization, implementation, invocation, etc.). Considering that we are dealing with large-scale programs (thousands of interrelated classes), we represent these relationships using structure filters, *i.e.*, queries. When a filter is applied on a particular class, related classes retain their original colors, while all other classes have the their color saturation reduced, as shown on Figure 3.

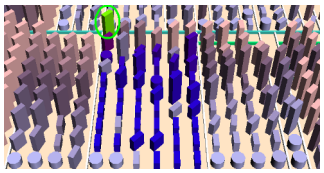

Figure 3. A relationship filter applied to the circled green class.

Semantic Information. Semantic information refers to application domain knowledge that is not explicitly represented in the source code. Some anomalies are defined/detected by taking into account such kind of knowledge (*e.g.*, the fact that a class plays a *controller* role combined with the other conditions can significantly improve the detection accuracy of *Blob*). During the detection using our visualization framework, if we need to determine if a class plays a certain role, the analyst can browse the code by mouse clicking on the corresponding class.

4. Anomaly Detection

This section describes our (semi-automatic) visualization-based approach to anomaly detection. First, we explain our general detection principle, from which we derive detection strategies for specific design anomalies. Then, we present an example of detection strategy for the *Blob*.

4.1. Detection Principle

Design anomalies targeted in this work take the form of either a single class having a set of undesired properties, or a micro-architecture involving a group of classes. In the latter case, one class often plays a primary role, while the others play supporting secondary roles. For example, in the *Blob* anomaly, the controller class plays the primary role, and the data classes play secondary roles. For all those design anomalies involving several classes, we found more efficient to search for primary-role classes first, and then focus on secondary-role classes. A reason for this is that primary-role classes often have abnormal metric values, and therefore are easier to detect in our visualization framework. Also, seeing micro-architecture design anomalies as scenarios played by primary and secondary roles greatly reduces the search space for the human analyst.

Based on that observation, we designed a general detection principle, from which we derive detection strategies for specific design anomalies. Our detection principle is summarized in Algorithm 1. Its first step sets a mapping between metric values and graphical attributes, using software metrics that are relevant to characterize the different roles of the anomaly. Then the analyst locates all candidate classes for the primary role of the anomaly. This can be done by looking at the program representation and searching for classes having a certain graphical appearance (corresponding to the mapping). Distribution filters can also be applied for this second step to visualize metric distributions and locate classes having abnormal values. All candidate classes for the primary role are marked by the detection tool. If the primary role must be supported by secondary roles (depending on the anomaly to detect), a subsequent step consists, for each candidate, in the application of relationship filters. The analyst considers then the graphical appearance and/or location of the related classes to decide if this satisfies the conditions of the considered anomaly. At

any moment and when semantic information is needed, the analyst can inspect names and/or source code of the located classes to ensure that they play the targeted roles.

Algorithm 1 Detection Principle

```
1: Set the mappings metric–graphical attribute
2: Consider the appearance of the classes in the whole program
   and/or apply the distribution filter
3: Mark every class that satisfies the primary role requirements
4: for each marked class c do
5:     Inspect the name and/or source code of c (if needed)
6:     if primary role must be supported by secondary roles then
7:         Apply relationship filters on c
8:         Consider the appearance and/or location of the classes
           related to c
9:     end if
10:    if all the above steps are positive then
11:        Save the occurrence
12:    end if
13: end for
```

4.2. Example of Anomaly Detection

The general principle defined above can be applied to detect a wide range of anomalies. In the current state of our work, we detect six anomalies: *Blob*, *Misplaced Class*, *Functional Decomposition*, *Swiss Army Knife*, *Divergent Change*, and *Shotgun Surgery* [4]. Due to lack of space, we present here only the detection of the *Blob* anomaly with a concrete detection example. The definition of the *Blob* anomaly can be mapped directly to its detection conditions. This anomaly involves two types of roles: the *God Class* that implements almost all the behavior and a set of *Data Classes* that mainly store the data. The detection is done in three steps: mapping, identification of *God Class* candidates, and inspection of related classes (*Data Class* candidates). Three metrics are used [2]. WMC , when abnormally high, indicates a potential *God Class*, and when very small, indicates potential *Data Classes*. LCOM5 is generally high for the *God Class* (implementation of independent functions). Finally DIT is supposed to be very low for both roles (very few ancestors for *God Class* and *Data Classes*). The *Blob* detection strategy is summarized in Algorithm 2.

An example of a *Blob* occurrence is shown in Figure 4. On the left, we see the result of the distribution filter for metric WMC. The circled black class was selected as a candidate controller class, because it has an extremely high WMC value[1] and is twisted (high LCOM5 value). In Figure 4 (right), the association 'out' relationship filter is applied on the candidate class. Most of the associated classes (*i.e.*, those that are still colored) are blue, small, and straight, meaning that they are *Data Classes*. The inspection of the source code of the controller class confirmed that it is actually a *Blob* occurrence. The *Blob*, *org.eclipse.swt.internal.win32.OS*, was detected in *RSSOwl*, an open-source RSS reader.

[1] When applying the distribution filter, classes with abnormal values for the considered metric are colored in red, independently from the mapping. Among them, the class with the largest value and not yet considered is colored in black to help the analyst navigating in the search space.

Algorithm 2 *Blob* Detection Strategy

```
1: Set the mappings metric–graphical attribute to:
   DIT-color, WMC-height, LCOM5-twist
2: Apply the distribution filter on WMC
3: Locate candidate classes with an extremely high WMC value
4: for each candidate class c do
5:     Inspect LCOM and DIT of c
       (must be a blue and fairly twisted box)
6:     If necessary, inspect the name and code of c
7:     Apply association 'out' filter on c
8:     Inspect classes associated with c
       (must be small (low WMC), straight (highly cohesive), and
       blue (low DIT))
9:     if all the above steps are positive then
10:        Save the occurrence of Blob
11:    end if
12: end for
```

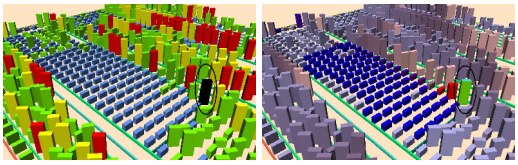

Figure 4. A *Blob*. (left) Box plot filter on WMC. (right) Association filter on the circled class.

5 Case Study

To evaluate our approach, we conducted a case study. We selected two software systems , *PMD 1.8* (286 classes) and *Xerces 1.0.1* (296 classes), on which subjects accomplished anomaly detection tasks using VERSO. For this study, we were interested in three design anomalies: *Blob*, *Functional Decomposition*, and *Swiss Army Knife*. We systematically performed a manual inspection of both software systems to build a list of all the occurrences of those anomalies in the systems, in order to compute the recall.

The evaluation was conducted with 15 subjects, graduate students with background in programming and software engineering. The subjects had to detect occurrences of the three design anomalies. They were separated in the two groups A and B. Subjects in group A used VERSO on *PMD* and manual inspection on *Xerces*; subjects in group B used manual inspection on *PMD* and VERSO on *Xerces*. The detection with VERSO had to be performed in at most 30

minutes per software system to detect occurrences of the three design anomaly types. There was no minimum time.

As mentioned in Section 2, analysts using VERSO detect and validate anomaly occurrences simultaneously. Thus, all identified anomalies are considered as true positives according to the analyst interpretation. In this context, the precision is always 100%. For this reason, we excluded precision from the evaluation criteria. Instead we investigated the variability of the number of occurrences detected between the subjects, and the recall based on the list of anomaly occurrences we manually found for each system.

Our first finding was that the variability between subjects, considering the number of occurrences detected, is less with VERSO than for manual inspection, although the averages were very close. Table 1 (left) shows the standard deviation of the number of occurrences detected for manual inspection and VERSO on both systems.

	Standard Deviation		Recall for VERSO	
	PMD (Man / VERSO)	*Xerces* (Man / VERSO)	*PMD*	*Xerces*
Blob	2.63 / 1.2	3.74 / 2.93	2 / 5 (40%)	5 / 13 (38%)
FD	4.56 / 2.55	3.01 / 2.54	12 / 16 (75%)	25 / 30 (83%)
SAK	4.84 / 0.87	2 / 1.25	0 / 1 (0%)	1 / 2 (50 %)

Table 1. (left) Standard deviation for the number of occurrences detected. (right) Recall.

Our second finding was related to the recall of the anomalies detected using VERSO. In our case, the recall is the proportion of all anomaly occurrences present in the systems that were detected with VERSO, taking the results of the score of the best subject. Table 1 (right) shows the recall for the three anomalies with both systems. Considering the limited time and the size of the program, the recall is very good for *Functional Decomposition* and average for *Blob*. This could be explained by the fact that *Functional Decomposition* is a single role anomaly and was probably easier to detect than a micro-architecture of several classes such as the *Blob*. The recall for the *Swiss Army Knife* is difficult to interpret because very few *Swiss Army Knife* occurrences were present in both systems.

6. Conclusion

Anomaly detection and correction is a concrete and effective way to improve the quality of software. This paper proposes a semi-automatic detection approach that combines automatic pre-processing and visual representation and analysis of data. Our approach is complementary to automatic approaches for anomalies whose detection requires knowledge that cannot be easily extracted from the code directly. The detection is seen as an inspection activity supported by a visualization tool that displays large programs (thousands of classes) and allows the analyst navigating at different levels of the code. More specifically, we detect occurrences of anomalies by viewing them as sets of classes playing roles in predefined scenarios. Primary roles are identified first, and starting from them, secondary roles are located. This strategy allows to reduce the search space which compensates for human intervention.

Although our case study showed interesting results, it revealed that there is room for improvement. The performance variability of subjects still needs improvement. To reduce this variability, we are defining features to better guide the analyst when exploring the search space. Combining our approach with an automatic one is another direction we are investigating. Three ways of combination are possible. First, we can use our tool to explore the results of automatic detection and accept/reject the detected occurrences. We can also use the two approaches in parallel, depending on the required knowledge to detect the anomalies. Finally, we can use stylized focus [3] to attract the attention of the analyst toward parts of the program where candidate occurrences, detected automatically, are present.

References

[1] W. Brown, R. Malveau, H. McCormick, III, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[2] S. Chidamber and C. Kemerer. A metric suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):293–318, 1994.

[3] F. Cole, D. DeCarlo, A. Finkelstein, K. Kin, K. Morley, and A. Santella. Directing gaze in 3D models with stylized focus. In *Eurographics Symp. on Rendering*, pages 377–387, 2006.

[4] K. Dhambri. Détection visuelle d'anomalies de conception dans les programmes orientés objets. Master's thesis, Dept. I.R.O., University of Montreal, Dec. 2007.

[5] N. Fenton and S. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, 1998.

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

[7] Y.-G. Gueheneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *Proc. OOPSLA*, pages 301–314, 2004.

[8] Y.-G. Gueheneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *Proc. WCRE*, pages 172–181, 2004.

[9] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proc. ASE*, pages 214–223, 2005.

[10] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proc. IEEE ICSM*, pages 350–359, 2004.

[11] R. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[12] N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *Proc. ASE*, pages 297–300, 2006.

[13] A. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.