Object-Oriented Design in Practice. Could/Should We Do Better ?

Radu Marinescu*

* Institute e-Austria Timişoara
radum@cs.utt.ro

Abstract

There is no perfect software design. Like all human activities, the process of designing software is error prone and object-oriented design makes no exception. The flaws of the design structure have a strong negative impact on quality attributes such as flexibility or maintainability. Thus, the identification and detection of these design problems is essential for the evaluation and improvement of software quality.

The goal of this paper is to show how structural flaws in object-oriented design can be quantified and identified using a metrics-based mechanism, called *detection strategy*, which allows us to express in a measurable manner violations of design rules and heuristics.

1 Design is Hard

As Opdyke pointed out: "design is hard" [5]; and object-oriented design in spite of all the optimistic claims is not a bit easier. Object-oriented programming supports essential software development goals like maintainability and reusability [4] through mechanisms like encapsulation of data, inheritance and dynamic binding. Yet in the beginning of object-orientation many software companies had the naïve dream that the use of objects will automatically increase the quality of their software and greatly decrease the time spent on development and maintenance. It was hoped that by introducing object-oriented mechanisms software systems will become more flexible, more extensible and understandable, and that they will therefore be easier to maintain.

Today, the industry is confronted with a large number of software systems in use, in the size of millions of lines of code. By their inherent size, complexity and development times, they have reached the suitable shape for object-orientation. Yet, most of these systems lack all of the aforementioned qualities: they are instead monolithic, inflexible and hard to extend.

Thus, we may state that object-oriented programming is a basic technology, that supports quality goals like maintainability and reusability but just knowing the syntax elements of an object-oriented language or the concepts involved in the object-oriented technology is far from being sufficient to produce good software. A good object-oriented design needs design rules and practices that must be known and used. Their violation will eventually have a strong impact on the quality attributes.

2 Make Design Quantifiable

The market forces emphasize more and more the need for quality in software system, but as De Marco points out: "you cannot control what you cannot measure"[1]. Thus the question that comes to our mind is: can we quantify by the interpretation of software measures the principles and rules of object-oriented design?

2.1 Software Measurement is also Hard

In the last decade software metrics became more and more an important means for assessing and controlling the quality of software systems in general and for object-oriented systems in particular. By

analyzing former approaches we find out that there is still a large gap between the use of individual measurements and the principles that rule the construction of object-oriented software. Current approaches do not provide a way to quantify such design principles and rules.

2.2 Detection Strategies

In this context, our work introduced a new mechanism, named *detection strategy*, for increasing the relevance and usability of metrics in object-oriented design by providing a higher-level (more abstract level) means for interpreting measurement results. A detection strategy is defined as the quantifiable expression of a rule by which design fragments that are conforming to that rule can be identified in the source-code. The main goal of a strategy is to provide the engineer with a mechanism that will allow him or her to work with metrics on a more abstract level, which is conceptually much closer to his real intentions in using metrics. Such rules may refer both to design recovery (*understanding of design*) and the identification of design flaws (*evaluation of design*).

2.3 Defining Detection Strategies

The starting point for this approach is an informal description of a rule related to the design. This is the rule that we try to quantify. In recent years we have often found various forms of descriptions for such rules, in the literature [6] [2] [3]. The approach consists of the following sequence of steps:

1. *Analysis of the Design-Rule.* After choosing the design-rule that should be quantified, the first step is to express the informal description of the rule in a quantitative manner. In other words, we have to describe how the rule is related to the design entities (e.g. for a rule describing a design flaw we may have class inflation, excessive method length) and the relationships among them (e.g. highly coupled subsystems). The result of this step is the definition of a concrete strategy for detecting the conforming entities based on the analysis of the informal description.

2. *Selection of Metrics.* Based on the quantitative description of the design rule, those metrics must be found or defined that are most suitable for the measurement of the characteristics of the sought design structure. At the end of this step, the detection strategy can be expressed as a specific combination of the selected metrics.

3. *Detection of Candidates.* The third step is to measure the system based on the defined detection strategy, using the chosen metrics. The result of this step is a list of design fragments that are supposed to be conforming to that design-rule. The candidates are filtered from the initial data set based on the data filters associated with each metric in the definition of the strategy (e.g `HigherThan`, `TopValues` etc.) as shown also in Figure 1.

4. *Examination of Candidates.* The last step is to examine the identified design fragments and to decide, based on the source-code and on other information sources, if the candidates are indeed what we were seeking or if the proposed detection strategy didn't really properly capture the initial (informal) design-rule.

2.3.1 An Example: Detecting Data-Classes

Let's assume that we want to find a set of classes that exhibit their data in the interface. We decided to use two metrics: the first one to count the number of public attributes (NOPA) and the other one to count the number of accessor-methods (NOAM). We decide that the classes we want to find are those with the most public data from the project, but that they should not have less than 3 public attributes or 5 accessor-methods. Therefore, the following detection rule (in SOD) is used:

```
DataClasses :=
    (NOPA, HigherThan(3) and NOPA, TopValues(10%)) or
    (NOAM, HigherThan(5) and NOAM, TopValues(10%))
```

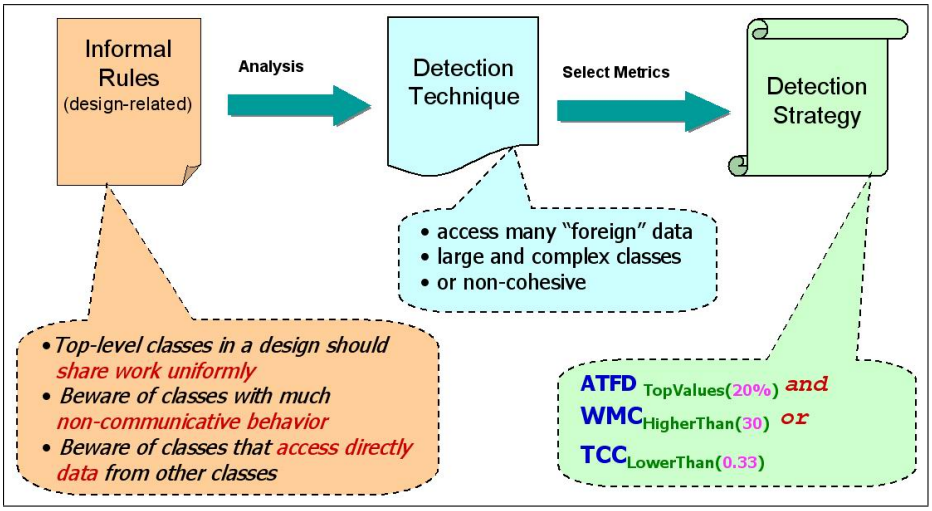

Figure 1: The steps for transforming an informal design rule in a detection strategy. In the bottom part of the picture the transformation is exemplified by using the description of the "Behavioral God-Class" design problem [6]

.

3 Summary

This paper briefly advocates the need for controlling the quality of design and introduces a new mechanism, i.e. *detection strategy*, that offers an encapsulation of metrics in a higher-level construct that permits a more meaningful interpretation and usage. For the first time, detection strategies become a mechanism for expressing rules related to the structure of code and design in terms of metrics. We defined not only the mechanism to objectify and quantify such rules, but we also provided a methodology for moving from informal descriptions of rules to detection strategies.

Although not presented in this paper, we succeeded in quantifying a large set of well-known design flaws described in the literature from "Feature Envy" [2] or "God Classes" [6] to places in the source code where a particular design pattern should have been applied.

The entire approach described above is strongly supported by tools. These tools have a high-level of language-independency, as they work on a unified meta-model for object-oriented languages. The provided tool support makes the approach scalable, and we proved that by analyzing real-world systems containing up to 1 MLOC.

References

[1] T. DeMarco. *Controlling Software Projects; Management, Measurement and Estimation.* Yourdon Press, New Jersey, 1982.

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

[3] R.C. Martin. *Design Principles and Patterns.* Object Mentor, http://www.objectmentor.com, 2000.

[4] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, 1988.

[5] W. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

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