Improving the automatic detection of design flaws in object-oriented software systems

Petru Florin Mihancea* Radu Marinescu*

* LOOSE Research Group, Research Institute e-Austria Timişoara
{pepy,radum}@cs.utt.ro

Abstract

In order to increase the flexibility and extensibility of an object-oriented software system, its structural flaws have to be detected. The practice shows that the automatic approach for design flaw detection based on metrics is not very accurate. While the choice of the used metrics can be easily argued, the values for their thresholds are usually chosen based only upon the designers experience. In this paper we present a method for tuning the values of the thresholds. Experimental results are also shown in order to prove the efficiency of our method.

1 Motivation

Software has to evolve. Although today it meets all the requirements imposed by an user, there will be a time when new features must be added to the program as a result of its environment evolution. Unfortunately, because of frequent and improper modifications made on a piece of software, its internal structure breaks making further modifications very hard to implement. In order to prevent such undesired situations, we must correct the software internal structure from time to time.

An important task in the correction process is *problem detection*. We have to find those design entities (i.e. classes, methods) that hinder the easy evolution of the software. In other words, we have to detect the entities affected by some *design flaws*. A common approach is manual inspection of the source code. Informal descriptions of many object-oriented design flaws can be found in literature [1]. Using these descriptions, we can manually detect all the entities affected by the described flaws. Obviously, the manual approach does not scale up for very large programs.

The author of [2] describes an automatic approach for problem detection. *Software metrics* represent the basic element of this method. A metric measures a particular property of a design entity. Abnormal values of a particular set of metrics can show that the measured entity is affected by a particular design flaw. Thus, the idea of the aforementioned approach is to quantify the informal description of a design flaw, obtaining a *detection strategy* for that flaw. A detection strategy specifies which metrics must be observed, which maximal or minimal value is accepted for each metric, and how the abnormal measurements are correlated in the context of the aimed design flaw.

By the fact that detection strategies can be executed automatically on a piece of software, the process of detecting design flaws has been speeded up, but the practice has also shown a problem: there are many situations in which design entities are incorrectly detected and/or design entities that should be detected are missed. There might be many causes for this, but the practice has also shown that the most frequent cause is represented by improper assignments for the threshold values of metrics. These assignments are very important because, as stated in [2], "the quality of a detection strategy strongly depends on the proper selection and parameterization of a filtering mechanism". In [2] a "methodology" for this operation is described, but it does not explain how the threshold values for the proposed detection strategies were established. So, in order to improve the precision of a detection strategy, a clear method for the assignment of threshold values for the metrics, is strongly needed.

2 The detection strategy tuning machine

In [5] we address the problem of detection strategy parameters. First, we have to understand what the *correctness* of a detection strategy means. Obviously a strategy is perfectly correct if it detects all design entities affected by the design flaw quantified by the strategy, and none other. We believe that a formal demonstration for the perfect correctness of a strategy is hardly possible to obtain because of the extraordinary variety of forms that a particular design problem can take. This statement goes together with Fowler's statement that "no set of metrics rivals informed human intuition" [1].

Unfortunately, an automatic approach for problem detection needs a quantified form of the design flaws. But because we can not surely obtain a perfect quantification, we have to rely on a good approximation. Thus, we will assume that a detection strategy is "correct enough" if it properly classifies a large set of positive and negative examples of the quantified flaw. A positive example of a flaw is a design entity affected by that particular flaw. In a similar way, a negative one is an entity which is not affected by that particular design flaw.

In this paper, because we are focused only on the parameters problem, we assume that the *skeleton* of a detection strategy properly quantifies most of the particularities of the observed design flaw. In other words, the metrics and their correlation are relevant in the context of the quantified design problem. As a result, obtaining the threshold values for a detection strategy is the problem of finding those numbers that assure the consistency of the detection strategy with the largest possible number of positive and negative examples of the quantified design problem. In this context, we introduce the concept of detection strategy tuning machine.

Definition 1 *The detection strategy tuning machine is a mechanism that helps finding the parameters of a detection strategy skeleton by automatically tuning their values in such a way that the resulting strategy is consistent with the largest possible number of positive and negative examples of the design problem addressed by the strategy.*

Figure 1 depicts the main components of the tuning machine and their flow of interaction. Next, we are going to introduce these components by following the main steps of the tuning process for a detection strategy.

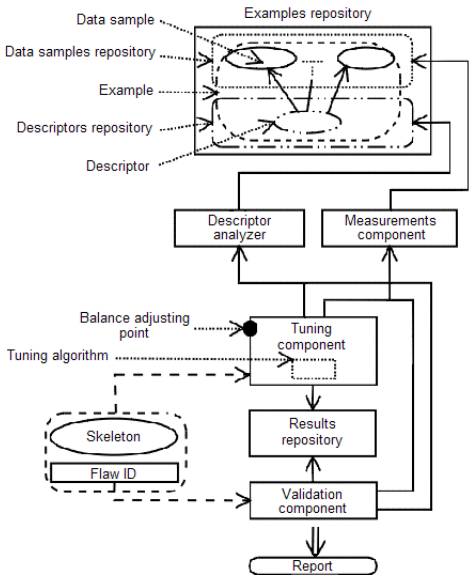

Figure 1: The meta-architecture of the tuning machine

Step 1. Collect the examples. In order to tune a detection strategy, we have to collect a set of positive and negative examples, based on the informal description of the flaw captured by the strategy to be tuned. All these examples will be stored in the *examples repository* in form of a collection of *data samples*, the atomic units of an example, and a *descriptor* that characterizes the example (i.e. which data samples compose the example, what design flaw it addresses, it is positive or negative, it will be used for tuning or for validation).

Step 2. Tuning. The skeleton of a strategy and an identifier for the design problem which is addressed by the strategy are the inputs for the *tuning component*. This component is the one which has to find the parameters using a tuning algorithm. By analyzing the inputs, the tuning component will ask the *descriptor analyzer* to prepare the set of examples used for tuning operation. When the tuning process is in progress, the values for some metrics will be needed. The tuning component will ask the *measurements component* to calculate these values. When the tuning process is finished the resulted detection strategy is inserted in the *results repository*.

Step 3. Validation. In conformity with Definition 1, the obtained tuned strategy will be consistent with the largest possible number of positive and negative examples from the tuning set. But, we also need to know how many positive examples from a particular set are missed (false negatives) and how many negative examples from the same set are viewed as positive (false positives) by the tuned

Table 1: The precision of the GodClasses detection strategy

Set type	Positive examples	Negative examples	Original parameters		Tuned parameters	
			true positives	false positives	true positives	false positives
Tuning	9	142	3	4	8	2
Validation	1	22	0	0	1	1

Table 2: The precision of the DataClasses detection strategy

Set type	Positive examples	Negative examples	Original parameters		Tuned parameters	
			true positives	false positives	true positives	false positives
Tuning	18	126	6	3	12	2
Validation	4	41	1	0	1	0

strategy. Using these measurements we can evaluate the precision of the resulted detection strategy. The *validation component* is the one responsible to evaluate the tuned strategy that addresses the design problem specified at its input. In the end, it will produce a report for the user. It is important to mention that an evaluation based on the set of examples that has been used for tuning is not sufficient. In order to obtain a relevant evaluation the validation component must also use a set composed with examples that were not seen by the tuning component. This is the validation set.

Step 4. Re-tuning. A final element of the machine is the *balance adjusting point*. There might be situations when the tuning set contains many conflicting positive and negative examples that can not be simultaneously correctly classified. This is a sign that something is not right with the skeleton. The problem in these situations is that the tuned strategy will miss many real flaws or it will introduce many false positives. From the problem detection process point of view, the first case is more dangerous. It is preferable to be detected as many real flaws as possible, but having an acceptable number of false positives. If it is necessary, the balance feature can be used in order to obtain a tuned strategy with a smaller number of false negatives. This is done by enforcing the correct classification of more positive examples and by accepting the necessary increase of the number of false positives.

3 Experiments and results

To validate our method we have implemented a prototype for the tuning machine and we have used it for the tuning of a couple of detection strategies. The tuning algorithm is a classic genetic algorithm [4]. It searches for the best parameters of a detection strategy by trying to minimize the overall penalty produced by the incorrect classification of some examples. At a normal balance the penalties for a false negative and for a false positive are equal. When they are not equal the machine will favor the elimination of the inconsistencies with higher penalty if it can obtain a smaller overall penalty.

To collect positive and negative examples for different design flaws we have selected 7 medium-size object-oriented software systems. We have also built a catalog of design problems containing the informal descriptions of a set of design flaws found in the state-of-the-art literature [1, 6]. The systems have been manually inspected by 11 persons which, based on the problem catalog, have indicated all the design problems found. Using their reports, we have constructed a set of positive and negative examples for the GodClass design flaw and another one for the DataClass design flaw (both defined in [1, 6]). We have split each set into two subsets: one has been used to tune the parameters of the aimed detection strategy skeleton, and the other has been used for validation. The results, obtained at a normal balance, are presented in Tables 1 and 2.

First, we will discuss the experiment for the GodClass detection strategy. It is easy to see that the strategy that uses the original parameters from [2] is not very precise: only 3 out of 9 positive examples from the tuning set are detected. Using the parameters provided by the detection strategy tuning machine, almost all the design problems from the tuning set are detected and the number of false positives is smaller. Of course, this improvement over the tuning set was expected: the detection

strategy tuning machine searches for those parameters which minimize the number of inconsistences. Now, let's take a look on the precision over the validation set which was not seen by the tuning algorithm. The usage of the initial parameters hinders the detection of the positive example. But if we use the new parameters this positive example is detected, and only one false positive appears. So, we can conclude that the precision of the strategy has been improved.

Next, we will discuss the experiment made for the DataClasses detection strategy. The precision of the initial strategy over the tuning set is not very good: only 6 out of 18 positive examples are detected. After the tuning process the precision increases and the number of false positives is smaller. Unfortunately, the improvement does not appear over the validation set. But neither a degradation! So, the new parameters are better. Which is the reason for these results? A manual analysis over the tuning set has shown very significant conflicts between positives and negatives examples. This conflicts do not admit to make the strategy consistent with some typical positive example because, if they are detected, the number of false positives explodes. This conclusion has been acknowledged by the machine, when we have tuned the strategy at a 0.9[1] balance. Now the dilemma comes. Should we use in practice the parameters obtained at a normal balance, or the parameters obtained at 0.9? The response is simple: if we want to be sure that we'll miss a few real design flaws we must use the second set of parameters. At the same time, we must be ready to manage a probably large number of false positives. If we do not want many false positives, we must use the first parameter set. Of course, we'll risk to miss some real DataClasses.

We can conclude that the skeleton of the DataClass detection strategy is not very good. There are some particularities of DataClass design flaw that are too generally captured by the skeleton. In order to increase the precision beyond the above limitations, the flaws of the skeleton must be handled.

4 Conclusions and future work

The detection strategy tuning machine came in as an efficient mechanism for optimizing the precision of detection strategies. The validity of this conclusion was shown not only by our experiments. Significant improvements were obtain by using the tuned strategies and other related observations in an large-scale project, where detection strategies were applied to detect design problems [3]. On the other hand, the machine can be used to "detect" flaws of the current detection strategies. If too many inconsistences remain after the tuning process, the skeleton reanalysis is mandatory.

In the coming time we want to apply the tuning process to further detection strategies. A key element in doing this is to collect a large enough number of positive and negative examples. At the same time, we must be prepared to correct potential flaws that could be found in some detection strategies structures. An automatic tool to construct the skeleton of a detection strategy is also considered.

References

[1] M. Fowler. *Refactoring. Improving the design of existing code*, Addison-Wesley, 1999.

[2] R. Marinescu. *Measurement and quality in object-oriented design*, Ph.D. thesis in the Faculty of Automatics and Computer Science of the "Politehnica" University of Timişoara, 2002.

[3] R. Marinescu. D. Ratiu. *Detection of design problems in an industrial environment*, 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing, Timişoara, Romania, October 1-4, 2003.

[4] Z. Michalewicz. *Genetic algorithms + Data structures = Evolution Programs*, 3rd edition, Springer, 1996.

[5] P. F. Mihancea. *Optimizarea detecţiei automate a carenţelor de proiectare în sistemele software orientate pe obiecte*, Diploma thesis in the Faculty of Automatics and Computer Science of the "Politehnica" University of Timişoara, 2003.

[6] A. J. Riel. *Object-oriented design heuristics*, Addison-Wesley, 1996.

[1] At this balance a false negative is nine times harder penalized than a false positive