Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems

Petru Florin Mihancea Radu Marinescu

LOOSE Research Group
"Politehnica" University of Timişoara, Romania
E-mail: {petrum, radum}@cs.utt.ro

Abstract

*In order to increase the maintainability and the flexibility of a software, its design and implementation quality must be properly assessed. For this purpose a large number of metrics and several higher-level mechanisms based on metrics are defined in literature. But the accuracy of these quantification means is heavily dependent on the proper selection of threshold values, which is oftentimes totally empirical and unreliable. In this paper we present a novel method for establishing proper threshold values for metrics-based rules used to detect design flaws in object-oriented systems. The method, metaphorically called "tuning machine", is based on inferring the threshold values based on a set of reference examples, manually classified in "flawed" respectively "healthy" design entities (*e.g.*, classes, methods). More precisely, the "tuning machine" searches, based on a genetic algorithm, for those thresholds which maximize the number of correctly classified entities. The paper also defines a repeatable process for collecting examples, and discusses the encouraging and intriguing results while applying the approach on two concrete metrics-based rules that quantify two well-known design flaws* i.e.*, "God Class" and "Data Class".*

Keywords: threshold, metrics, design flaws, object-oriented design

1. Motivation

1.1. Introduction

Not only bugs cost! The "errors" of the internal organization of an object-oriented software system have a negative impact on important quality factors, such as its maintainability. Thus, design flaws [9] are going to generate high maintenance costs, and therefore hinder the evolution of the system. It is important to emphasize that design flaws are the result of evolution itself, more precisely of the "software aging" phenomenon [14]. The only thing we can do in order to slow down the aging process and thus to preserve the program technical and economical value is to "correct" from time to time those design entities (e.g., classes, methods) affected by structural errors. The question is how can we find them?

The problem of design flaws detection is addressed in [10] where the author proposes the usage of software metrics in order to quantify the deviations from design rules or heuristics which characterize a good object-oriented design. Using a set of software metrics, a filter for each metric denoting its abnormal values (e.g., HigherThan(5), LowerThan(2)), and simple composition operators (e.g., *and, or*) which combine the metric-filter pairs, the informal descriptions of design flaws like those presented in [4] or derived from [16] are quantified in form of a detection strategy.

As a simple example, let's consider that we want to detect classes that should be split because they represent improper abstractions. A detection strategy which expresses this design flaw is presented in Equation 1.

$$ToSplit(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (TCC(C), LowerThan(0.2)) \wedge \\ (NOM(C), HigherThan(7)) \end{array} \right. \quad (1)$$

The first elementary term of the strategy (TCC(C), LowerThan(0.2)) selects all the classes from the system having a low cohesion between their methods (TCC = Tight Class Cohesion [2]). The second elementary term selects all the classes with more than 7 methods (NOM = Number of Methods). The final result is computed by the *and* composition operator which acts as an intersection set operator. Thus, the classes having both characteristics will be selected for splitting.

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

1.2. Problem statement

As a result of [10], [11], [15] detection strategies seem to be an efficient mechanism for automatic detection of design flaws. But in order to quantify a design flaw in the form of a detection strategy two questions must be answered:

- What set of metrics should be used and how should they be combined in order to quantify a particular design flaw and why?
- Which are the proper threshold values for these metrics and why?

Without a clear answer for both questions it is hard to believe that the results of an experiment made in order to validate the efficiency of a detection strategy is not affected by a "fishing for results" approach. In other words it is hard to believe that the conclusions are not dependent on that particular experiment.

While the first question can be answered starting with the Goal-Question-Metric paradigm [1] [20], answering the second question is much more delicate. Following the aforementioned detection strategy example, which is the reason to consider that a value for the TCC metric lower than 0.2 is abnormal in the context of the quantified design flaw? Why not lower than 0.21 or 0.19?

In [10] it is recognized that the proper selection of threshold values has "a decisive influence on the accuracy of a detection strategy". Unfortunately, no methodology is provided in order to resolve this problem. The clear reason of the selection of particular threshold values for the metrics used by a detection strategy usually remains a mystery.

1.3. Organization

Through the rest of this paper we propose a generic method, named *tuning machine*, which can be used to establish the proper threshold values for a detection strategy. Section 2 presents this approach. In order to validate our method we developed a Java prototype to automate the approach. Section 3 briefly presents this prototype and discusses our experiments and results. In the end we present several related works, we discuss the advantages and disadvantages of our approach and finally we draw some conclusions.

2. The detection strategy tuning machine

2.1. Surrounding the problem

M.Fowler et al. state in [4] that "no set of metrics rivals informed human intuition". Similarly we state that no detection strategy which quantifies a design flaw can rival informed human intuition. But because an automatic approach for design flaws detection is strongly needed when a very large software system is to be analyzed, we have to rely on some "good quantifications".

A detection strategy can be viewed as a model of the quantified design flaw inferred by its author from a set of particular examples or instances and consistent with those instances. But can we say that such a model is absolutely correct? Can we say that it is going to be consistent with any other possible example? Well, we can't. We can only say that it must be approximately correct if the model is consistent with a sufficiently large set of instances. As a result, if a detection strategy is consistent with a large set of available examples it must be good enough for our problem detection goals.

2.2. Planning the attack

In order to explain how we intend to find the threshold values of a detection strategy we have to clarify some notions which are going to be frequently used through the rest of the paper.

Definition 1 *A positive example of a design flaw is a design entity (e.g., a class) affected by that particular flaw.*

Similarly, a negative example of a design flaw represents a design entity which is not affected by that particular flaw.

Definition 2 *A false negative for a detection strategy is a positive example of the quantified design flaw which is not identified as such by the strategy. We say that the strategy is not consistent with that positive example.*

Similarly, a false positive for a detection strategy is a negative example of the quantified flaw which is identified by the strategy as being a positive example. We say that the strategy is not consistent with that negative example.

In the context of the above definitions and following the discussion from the previous section we state that in order to find the proper threshold values used by a detection strategy we have to find those thresholds which ensure the consistency of the strategy with the largest number of positive and negative available examples of the quantified design flaw.

If this number is large enough we can say that the detection strategy which uses these new threshold values is a good enough quantification for the corresponding design flaw. Otherwise, we can say that the strategy does not properly capture the particular characteristics of the targeted design flaw. In other words, the *skeleton* of that detection strategy, representing the strategy expression without the threshold values, is not relevant for what was intended to be quantified.

Now, we can define the tuning machine approach.

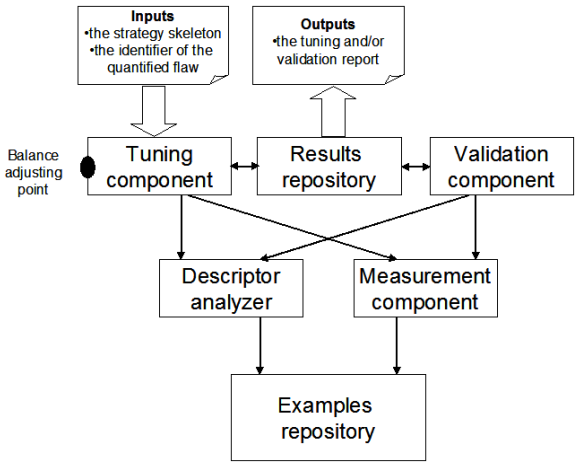

Figure 1. The meta-architecture of the tuning machine

Definition 3 *The detection strategy tuning machine describes a generic methodology which helps to find the proper threshold values of a detection strategy and to estimate its overall accuracy by tuning the thresholds in such a way that the resulting tuned strategy is consistent with the largest possible number of positive and negative examples of the design flaw quantified by that strategy.*

The first thing that we have to emphasize is that in this paper we describe a methodology. The "tuning machine" is only a metaphor for the approach. On the other hand, as resulting from the above definition, the tuning machine is independent on the flaw quantified by the strategy. All it knows is that a given set of examples is representative for what a given strategy quantifies and that it has to find those thresholds which maximize the consistency of that strategy over the specified set. At the same time, nothing is said about the algorithm used to find these thresholds. Thus, it is useful to describe the methodology in terms of a machine abstract architecture.

2.3. Attacking the problem

In this section, we are going to present the main steps which must be followed in order to tune a detection strategy. The description is made using the tuning machine components, each component being described from the statical and dynamical point of view. Figure 1 shows all the components of the machine and their way of interaction.

Step 1. Collect the examples. In order to tune a detection strategy, we have to collect a sufficiently large set of positive and negative examples, based on the informal description of the design flaw quantified by the strategy to be tuned. All these examples will be stored in the *examples repository*. The actual representation of an example is not important in the context of this paper. The only thing that we have to mention is that each example must be characterized by a *descriptor*. It has to specify at least what design flaw is addressed by the example and if it is a positive or a negative one.

Step 2. Tuning & validation. The skeleton of the strategy and an identifier for the design problem which is addressed by that strategy are the inputs for the *tuning component*. This component is the one which has to find the thresholds applying a *tuning algorithm*. First, using the flaw identifier, the tuning component asks the *descriptor analyzer* to prepare the set of examples used for the tuning operation. Next, the algorithm can start. While it is in progress, the values of some metrics used in the skeleton of the strategy which is being tuned are needed. The tuning component asks the *measurement component* to calculate all the metric values it needs. When the algorithm is finished the detection strategy obtained filling the skeleton with the identified thresholds is inserted in the *results repository*.

But tuning is not all. In conformity with Definition 3, 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 strategy. Using these measurements we can evaluate the accuracy of the resulted detection strategy and this is the responsibility of the *validation component*. Of course, an evaluation based on the set of examples that were already 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. All the sets of examples used by the validation component are constructed by the *descriptor analyzer* while the metrics values needed during the validation are calculated by the *measurement component*.

After the tuning and the validation sub steps a report is produced for the user in order to analyze the final results.

It is important to mention that the tuning and the validation sub-steps can be repeated multiple times when a cross validation technique is applied. Because of that the results repository must be seen in accordance with the repository/blackboard architectural style described in [19].

Step 3. Try a trade-off. 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. In other words there isn't any set of thresholds for the given strategy skeleton which simultaneously ensure the consistency of the strategy with these examples. Clearly this is a sign that something is not right with the

skeleton. The problem in these situations is that the obtained 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 detect almost all the real flaws, although an acceptable number of false positives is introduced.

The balance feature can be used in order to obtain a tuned strategy with a smaller number of false negatives. Through this adjusting point we can make a deal with the tuning algorithm before repeating the second step: we request the correct classification of more positive examples but we also have to accept the minimal necessary increase of the number of false positives. We can repeatedly search for an acceptable trade-off, but if we can't find a fair one we have to give up and correct the skeleton of the strategy.

3. Evaluation

3.1. The prototype

To validate our method we implemented the tuning machine in form of a Java program. Because the heart of the approach is the tuning component we are going to briefly discuss its implementation. All the implementation details can be found in [13].

In order to find the threshold values of a detection strategy the tuning component uses a genetic algorithm [12]. A potential solution or chromosome encodes one threshold for each elementary term of the strategy to be tuned (i.e, one for (TCC(C), LowerThan(_)) and one for (NOM(C), HigherThan(_))). The algorithm searches for those potential solutions which minimize the following fitness function:

$$f(X) = A * b * Fn\_No(X) + A * (1 - b) * Fp\_No(X) + 1 \quad (2)$$

where:

- X is the potential solution
- Fn_No(X) is the number of false negatives obtained when the tuned strategy is applied on the tuning set using the threshold values encoded in X
- Fp_No(X) is the number of false positives obtained when the tuned strategy is applied on the tuning set using the threshold values encoded in X
- A is a system parameter called *penalty amplitude* and it is set by the user. The parameter is a positive integer number and it helps to adjust the value of the penalty for one inconsistency produced by X
- b is a system parameter called *balance*. It is a number between 0 and 1 and it implements the balance adjusting point of the tuning component. When b is 0.5 a false negative and a false positive are penalized with the same score meaning that no trade-off is accepted (see Section 2.3)

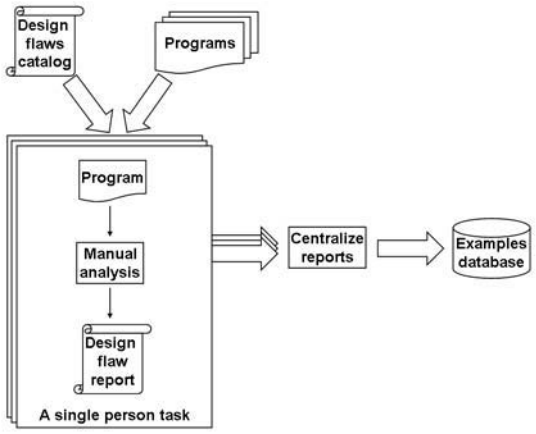

Figure 2. Examples acquisition process

Due to space limitations of this paper we will not discuss other characteristics of the algorithm such as the solution representation, selection model, etc..

3.2. Collecting examples

A special attention was given to the examples acquisition. The applied process is summarized in Figure 2.

First, we created a design flaws catalog containing informal description of many object-oriented design problems collected from literature [4], [16]. We also selected 7 object-oriented software systems written in Java and C++ each of them having approximately 30 classes. These systems were developed during the software engineering course at "Politehnica" University of Timişoara by students with little experience in object-oriented design. These programs were then distributed to 11 volunteers, senior students which had followed an advanced object-oriented design course. Each of them manually analyzed one system, and wrote a design flaws report based on our design flaws catalog. A report contained all the design entities which were found as being flawed and a short motivation for each reported entity. In the end, we centralized the reports, we mediated some inter-report conflicts, and we created an examples database. All design entities (e.g., classes) that were reported as being affected by a particular design flaw (e.g., they are DataClasses [4]) were marked in the database as positive examples for that flaw. The remaining entities (e.g., classes which are not DataClasses) were marked as negative examples of that flaw.

It is important to observe that our volunteers didn't know absolutely anything about how we quantified different design flaws (e.g., what software metrics we used) because

their manual analysis was based on the informal descriptions presented in the design flaws catalog.

3.3. The detection strategies

Before the discussion about the experiments we made it is important to present the detection strategies which were tuned using our tuning machine prototype. We are going to present their original form introduced in [9].

GodClass detection strategy. The GodClass design flaw affects those classes which tend to centralize the intelligence of a software system [9]. This is a deviation from an object-oriented design heuristic which states that the intelligence of a system has to be uniformly distributed among the top-level classes of the system design [16]. The quantified form of this design flaw is presented in Equation 3. Is essence, the GodClass detection strategy looks for classes which use data from the classes around them while they are complex or have low cohesion between their methods. Next, we are going to introduce the software metrics used to quantify the aforementioned "symptoms".

- *Access To Foreign Data.* ATFD represents the number of external classes from which a given class accesses attributes, directly or via accessor methods (get/set methods) [8].
- *Weighted Method Count.* WMC is the sum of the static complexity of all methods in a class [3].
- *Tight Class Cohesion.* TCC is defined as the relative number of directly connected methods. Two methods are directly connected if they access a common instance variable of their class [2].

DataClass detection strategy. The DataClass design flaw is introduced in [4]. A class affected by this flaw is nothing else than a "dumb data holder" having "fields, getting and setting methods for the fields and nothing else". The quantified form of this flaw is presented in Equation 4. This detection strategy searches for "lightweight" classes which provides almost no functionality through their interfaces while having many accessor methods (get/set methods) or data fields in their interfaces. The metrics used by this detection strategy are:

- *Weight Of a Class.* WOC represents the number of non-accessor, non-inherited methods in the interface of the measured class divided by the total number of interface non-inherited members [8].
- *Number of Public Attributes.* NOPA is defined as the number of non-inherited attributes that belong to the interface of the class [8].
- *Number Of Accessor Methods.* NOAM is defined as the number of the non-inherited accessor methods declared in the interface of a class [8].

3.4. Experiments

After we collected positive and negative examples for the GodClass and DataClass design flaws we applied our approach in order to find the threshold values for the corresponding detection strategies presented in Equation 3 and 4. For each strategy, we are going to present its accuracy before and after the tuning process and we are going to discuss the new thresholds. The strategies were tuned using the following configuration for the genetic algorithm:

- Population size: 60
- Number of generations: 100
- Crossover probability: 0.5
- Mutation probability: 0.1
- Representation: Floating-point [12]
- Selection strategy: Wheel model combined with elitist model [12]
- Crossover strategy: Heuristic [12]
- Mutation strategy: Uniform mutation [12]
- Penalty amplitude (A): 100
- Balance (b): 0.5 for GodClass and 0.55 for DataClass

We recognize that the number of examples we used is relatively far from being a sufficient one. Thus, the results have to be treated accordingly. In spite this, they demonstrate that our approach is a promising one.

3.4.1. Tuning GodClass detection strategy In Table 1 the composition of the tuning set and of the validation set is presented. The table also shows the accuracy of the strategy over these two sets using the original thresholds and those obtained by the tuning machine.

It is easy to see that the strategy that uses the original thresholds is not very accurate: only 3 (9 positive examples - 6 false negatives) positive examples from the tuning set are detected. Using the thresholds 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 thresholds which minimize the number of inconsistencies. 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 threshold hinders the detection of the positive example. But if we use the new one this positive example is detected, and only one false positive appears.

$$GodClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (ATFD(C), TopValues(20\%)) \wedge (ATFD(C), HigherThan(4)) \wedge \\ ((WMC(C), HigherThan(20)) \vee (TCC(C), LowerThan(0.33))) \end{array} \right. \quad (3)$$

$$DataClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (WOC(C), BottomValues(33\%)) \wedge (WOC(C), LowerThan(0.33)) \wedge \\ ((NOPA(C), HigherThan(5)) \vee (NOAM(C), HigherThan(5))) \end{array} \right. \quad (4)$$

Set type	Positive examples	Negative examples	Using original thresholds		Using tuned thresholds	
			False negatives	False positives	False negatives	False positives
Tuning	9	142	6	4	1	2
Validation	1	22	1	0	0	1

Table 1. The accuracy of the GodClass detection strategy

We cannot conclude that the accuracy of the strategy over the validation set have been improved because the number of inconsistencies is constant (although it is more important to eliminate a false negative and to introduce a false positive than the opposite situation). But it is important to observe that the inferred thresholds do not alter the strategy accuracy. So, these new thresholds inferred from the tuning set are not only a fortunate "accident".

For a better argumentation of the above statement let's take a look at the concrete threshold values we obtained. Are they semantically relevant? In other words, we want to see if the new threshold value for each elementary term of the strategy is compatible with the presumption about the quantified flaw which decided the usage of that term in the strategy. For example, the presumption which decided the usage of the (WMC(C),HigherThan(_)) term in the GodClass detection strategy was that a GodClass could be complex as we explained in Section 3.3. A semantically relevant threshold for this term has to be compatible with this presumption. A value of 40 or 50 is compatible but a value of 1 or 5 is not because almost every classes of a system can have such a value for the WMC metric and so the aforementioned term of the strategy does not help to select only the complex classes as it was intended to. In the second case it is most likely that the tuning machine has captured from the tuning set a fortunate but not a general particularity of the GodClass design flaw. This phenomenon is called overfitting [17].

In Figure 3 we present the threshold values obtained for each elementary term the GodClass detection strategy. Because of plateaux in the parametric space [12] we obtained more than one set of thresholds which produce exactly the same precision for the strategy as it has been presented in Table 1. We assign a unique number for all these sets which

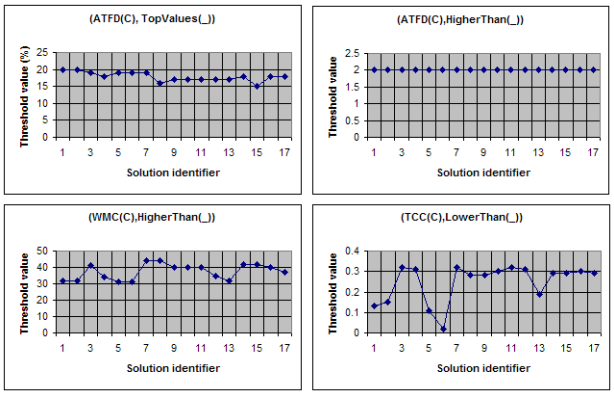

Figure 3. Thresholds for the GodClass detection strategy. The value on the X axis identify the solution from which the threshold is part and the value on the Y axis presents the concrete value of the threshold

reflects the order in which they were provided by the machine. Thus, each graphic from Figure 3 presents on the Y axis the thresholds value of the corresponding term of the strategy, while the X axis represents the unique identifier of the set from which that threshold is part. In this manner we can also see how a particular threshold varies from one set of thresholds to another.

The first thing we notice from Figure 3 is that the threshold of the (ATFD(C), HigherThan(_)) term of the strategy is the same in every solution and more important, its value (2) is semantically relevant. If a class breaks the encapsulation of some few other classes this action enables the suspicion that this class does something that should be done

by the accessed classes from the good object-oriented design point of view. In other words, our class has absorbed the intelligence of the accessed one.

The threshold of the (ATFD(C), TopValues(_)) term of the strategy varies between 15% and 20% from one solution to another. Any value in this interval is semantically relevant for what it is intended to be captured by this term of the strategy. As we mentioned earlier, GodClasses tend to centralize the intelligence of a software system. Thus, the number of classes affected by this flaw in a system is expected to be small relative to the total number of classes. So, it is reasonable to search for GodClasses between those classes having the first 15-20% top values for the ATFD software metric.

The threshold obtained for the (WMC(C), HigherThan(_)) term of the strategy it is also semantically relevant. The reason of using this term in the GodClass detection strategy is that a GodClass tends to centralize the intelligence of the system. Because of that it could be complex. A value for the WMC metric higher than 31 (the lower limit for the variation of the threshold) makes sense when we speak about the complex classes of a system.

The problem appears in the case of the (TCC(C), LowerThan(_)) term. It was included in the strategy because, as a result of [16], a GodClass may have a low cohesion between its methods. According with the interpretation model of the TCC metric [2] a value lower than 0.33 (the upper limit for the variation of the term threshold) expresses a low cohesion between the methods of the measured class. Unfortunately, in the 6th set of thresholds provided by the machine the value of the threshold is 0.02. Because the lowest possible value for the TCC metric is 0 it is hard to believe that this term can be responsible for detecting classes having "low" cohesion when this threshold value is used. In addition, because all the sets of thresholds provided by the machine produce the same accuracy for the strategy as it was shown in Table 1 it seems that it makes no difference if the term is included or not in the strategy. Well, this is not true.

In Figure 3 we noticed that the thresholds for the (WMC(C), HigherThan(_)) and (TCC(C), LowerThan(_)) terms varies together. This observation is demonstrated by the scatter diagram presented in Figure 4 where we can see that the thresholds for these two terms concentrate in two clusters.

The second cluster presents semantically relevant thresholds for both elementary terms of the strategy and it is clear that it can capture the true particularities of the GodClass design flaw. Only the first cluster presents the aforementioned problem and it is most likely affected by the overfitting phenomenon. If we provide more examples to the tuning machine this first cluster will probably disappear.

As a conclusion we can say that our tuning machine managed to find a couple of sets of relevant threshold values for the GodClass detection strategy. On the other hand, these new thresholds at least conserved the accuracy of the strategy. In this context we state that the tuning machine seems to be a fruitful approach for the threshold values problem of a detection strategy and a promising way to optimize the accuracy of a detection strategy.

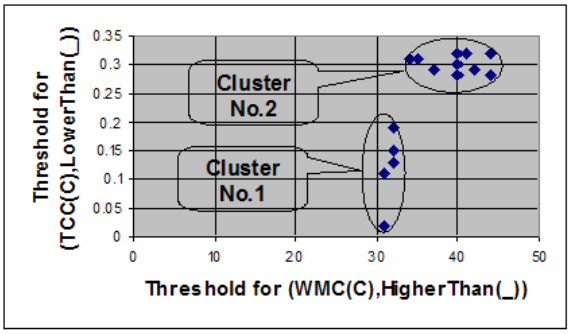

Figure 4. Threshold clusters

3.4.2. Tuning DataClass detection strategy Following a similar scenario like in the case of GodClass detection strategy we applied the tuning machine approach for the DataClass detection strategy.

When the strategy was tuned without any tradeoff (b = 0.5) meaning that the false negative and false positive inconsistencies are penalized in exactly the same way we obtained two categories of best solutions presenting the same total number of inconsistencies (8) : one having 7 false negatives and 1 false positive over the tuning set, and one having 6 respectively 2. In order to simplify the discussion and because we prefer a smaller number of false negatives we accepted a small tradeoff by tuning the strategy with a 0.55 value for the b parameter (balance) of the tuning algorithm. In this way the first category of solutions disappears.

In Table 2 the composition of the tuning set and of the validation set is presented. It is also shown the accuracy of the strategy over these two sets using the original thresholds and the one obtained by the tuning machine.

The precision of the initial strategy over the tuning set is not very good: only 6 (18 positive examples - 12 false negatives) are detected. Using the thresholds provided by the tuning machine the accuracy of the strategy has increased because the number of false negatives and false positives is smaller. Unfortunately, the improvement does not appear over the validation set. But neither does a degradation! Thus, the new thresholds are better than the original ones from the tuning set point of view and are at least as better as the original ones from the validation set point of view.

But why didn't we obtain better accuracy? First, let's

Set type	Positive examples	Negative examples	Using original thresholds		Using tuned thresholds	
			False negatives	False positives	False negatives	False positives
Tuning	18	126	12	3	6	2
Validation	4	41	3	0	3	0

Table 2. The accuracy of the DataClass detection strategy

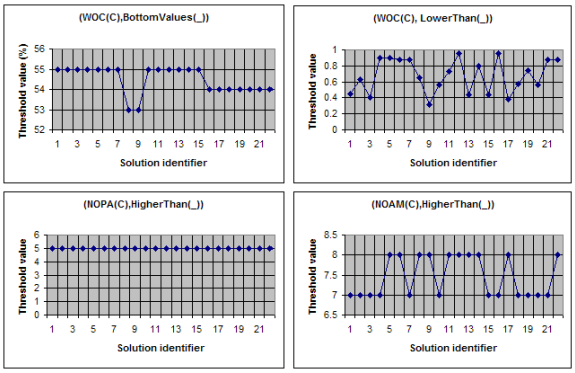

Figure 5. Thresholds for the DataClass detection strategy

take a look at the concrete thresholds established by the machine which are presented in Figure 5 in exactly the same way like in the case of the previous experiment.

We can see that the thresholds of three elementary terms are stable and their values are also semantically relevant for their corresponding terms of strategy. Without entering into details, a class having more than 5 public fields and/or more than 7-8 accessor methods can be suspected as having many fields and/or accessor methods in its interface.

The problem is with the threshold of the (WOC(C),LowerThan(_)) term because it seems to vary randomly while the accuracy of the strategy is the same for all the sets of thresholds provided by the machine. This suggests that this term is useless and we can safely delete it from the strategy. For example, in the 12th solution this threshold is 0.95. According to the interpretation model of the WOC metric [8] a class presenting such a value for this metric provides a lot of functionality (1 is the maximum value). Thus, a 0.95 threshold is not semantically relevant because the (WOC(C),LowerThan(_)) term should select all the "lightweight" classes from the system while a value lower that 0.95 for the WOC metric does not reflect this aspect. So, the strategy accuracy discussed in Table 2 is not influenced by this term at all, and all the "work" is done by the other terms of the strategy. As a final result, it is very likely that we have found a "flaw" in the skeleton of the strategy.

We continued our experiment by tuning the DataClass detection strategy at a higher balance (b = 0.9) accepting a more radical trade-off i.e. a false negative is nine times harder penalized than a false positive. The threshold we obtained for each term vary in small intervals but they are semantically relevant in the context of their corresponding elementary term. As we expected the number of false negatives reduced over the tuning set (1) and over the validation set (2). Unfortunately, the total number of false positives increased dramatically: 19 over the tuning set respectively 0 over the validation set. Remember that the number of positive examples contained by the tuning set is 18 and we obtained 19 false positives over this set! Thus, the tradeoff may be too radical.

Next, we manually analyzed the negative examples detected as false positive by the strategy tuned at a balance of 0.9 and the positive examples detected as false negatives by the strategy tuned at a balance of 0.55. In this manner we easily identified the problems responsible for the low accuracy of this detection strategy.

All these classes present a similar number of public attributes and/or accessor methods. The only difference is that the positive examples are very simple while the negative ones are very complex or inherit functionality from a super class (in many cases the super class is a library one). An OO designer may argue that some of these negative examples are in fact DataClasses or a form of DataClasses but they are clearly not totally conformant with Fowler's description of this flaw which states that a DataClass has "fields, getting and setting methods for the fields and nothing else" [4]. Although arguable, we will consider Fowler's definition as it is.

The problem is that all these classes have a similar relatively low value for the WOC metric. The interpretation model of this metric states that a class with a low value for WOC does not provide almost any functionality [8]. But our negative examples demonstrate the contrary: a class can provide much functionality (e.g., in one non accessor or inherited public method) while having a small value for the WOC metric. So, we can say that this metric does not really capture what it claims or at least its usage in the context of DataClass design flaw detection is strongly limited.

As a conclusion, the tuning machine managed to find a couple of sets of threshold values for the DataClass detection strategy which partially improved its accuracy. In the same time the machine raised an serious alarm signal that the strategy accuracy is strongly limited. If we use the

threshold values obtained at a balance of 0.55 we may miss too many DataClasses and if we use the thresholds obtained at a balance of 0.9 we may incorrectly detect too many non DataClasses. Analyzing the conflicting examples which were very simple to identify using our Java implementation of the tuning machine we also found the "flaw" of the strategy which has to be corrected in order to increase the strategy precision.

4. Pros and cons

Although based on our experiments we believe that the tuning machine is a good approach for establishing the threshold values for a detection strategy, we also identified some limitations.

One of the major problems of our approach is that in order to find the threshold values for a detection strategy we have to collect a set of positive and negative examples for the targeted design flaw. The only thing we can say is that it may be hard and time consuming but it is not impossible. Almost every day we find design flaws in our students' projects. Almost every company has to maintain its programs. So, if one really wants to do it, s/he can.

It is also important to mention that collecting positive and negative examples of design flaws can have unpredicted benefits. For examples, it is possible that the threshold values of a detection strategy and even its skeleton may depend on the size and on the programming language of the system which is going to be analyzed. Collecting examples for different classes of systems may produce in time a detection strategies catalog with specific quantifications of design flaws for different programming languages, for different sizes of systems, and so one. Not to mention that a company will be able to calibrate the strategies for its specific environment.

Another problem is that as it happened in our experiments, we might find two or more sets of threshold values, presenting small differences, but the same accuracy on the tuning and validation sets of examples. Which one to use in order to detect that design flaw in a software system? A possible solution is to use all of them obtaining a *multiple-thresholds detection strategy*. The idea is to repeat the analysis for each set of thresholds. All the entities detected as being flawed for at least one time will be reported and they can be ordered depending on how many threshold sets have detected them.

5. Related work

An old proverb says "use numbers, but never trust them!". This is particularly important when we search for software metrics thresholds. Nobody has demonstrated the existence of a certain threshold value for a particular metric. All we can do is to "guess" based on our experience giving birth to heuristic thresholds.

Lorenz and Kidd defined in [7] threshold values for many design software metrics. Unfortunately, these thresholds are very hard to use in order to establish the thresholds of a detection strategy for the following two reasons.

First, the authors do not say too much about the design problem context in which that threshold should be used. A particular metric could be used to quantify different design flaws but it is not clear in the context of which one its heuristic threshold should be used. From our point of view using the same threshold is not a good option because the threshold may be dependent on the concrete design flaw.

Second, the identified thresholds reflect the authors's experience with Smalltalk and C++ projects. So, using their thresholds in another development environment could be an inappropriate decision.

The detection strategy tuning machine method avoids these problems because it is based on a more abstract heuristic: a set of thresholds of a detection strategy is better than another one if the accuracy of the strategy is higher when it uses the first set of thresholds. The thresholds are identified for a particular detection strategy using a concrete set of examples of the quantified design flaw. If the examples are collected from a specific development environment the results can capture its particularities.

French describes in [5] a method for establishing software metric thresholds which can be used to focus the maintenance activity to truly problematic code. The problem of this approach is that almost nothing is said about the design problems the identified code has. It is important to know the design flaws we are dealing with because the necessary reorganization actions can be quickly identified in terms of code refactorings [4] or correction strategies [6]. This is one of the main benefits of a detection strategy.

An interesting alternative to deal with thresholds can be found in [18]. The basic idea is to map precise threshold values into fuzzy ones. Extending this approach to the thresholds of a detection strategy will improve its expressiveness because it will be defined in terms of "high complex classes" or "classes having many public attributes". This may be a very interesting research direction.

Another interesting approach is to enhance the problem detection process by combining detection strategies applied on a single version with additional information extracted from multiple versions of the analyzed system [15]. The approach is not intended to help finding the proper threshold values of a detection strategy, but it can improve the accuracy of the problem detection process by narrowing the focus of maintenance activity on the most dangerous flawed entities. Like any other design analysis method based on software metrics the success of this approach depends on some properly selected thresholds.

6. Conclusions and future work

We have presented in this paper a generic method which addresses the problem of threshold values that should be used by a detection strategy. The basic idea is to infer these thresholds from a given set of positive and negative examples of the design flaw quantified by that strategy.

We also presented two experiments in which we applied the method for two well known detection strategies. We obtained promising results which showed that our method:

- Can find the proper threshold values that should be used by a detection strategy and in this context can improve the accuracy of that detection strategy
- Can help to identify "flaws" in the quantification of a design flaw in form of a detection strategy

The major drawback of the approach is that in order to tune a strategy a sufficient large set of examples has to be constructed while this operation is not a very easy one. But as we emphasize in another section, collecting examples from a specific development environment and tuning a strategy using these examples can generate in time great benefits for a re-engineering process developed in the same environment.

Our future work will be focused on the following directions:

- We are going to increase the relevance of our aforementioned experiments by collecting more positive and negative examples for the GodClass and DataClass design flaws
- We are going to collect examples for other design flaws such as ShotgunSurgery [4], RefusedBequest [4], and many other and we are going to apply our method in order to find the threshold values for the corresponding detection strategies defined in [9]
- We also consider extracting the entire strategy, not only the thresholds, from a given set of examples. The method will be probably based on a decision tree learning algorithm [17]

Acknowledgments. This work was partially supported by the Austrian Ministry BMBWK under Project No. GZ 45.527/1-VI/B/7a/02. We also want to thank the anonymous reviewers of late breaking paper section of Metrics 2004.

References

[1] V. Basili and D. Rombach. The TAME project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1988.

[2] J. Bieman and B. Kang. Cohesion and Reuse in an Object-Oriented System. In *Proceedings of The 1995 Symposium on Software Reusability*, pages 259–262. ACM Press, 1995.

[3] S. Chidamber and C. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

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

[5] V. French. Establishing Software Metric Thresholds. In *Proceedings of The 9th International Workshop on Software Measurement*, 1999.

[6] D. Iulian and A. Trifu. Strategy Based Elimination of Design Flaws in Object-Oriented Systems. In *Proceedings of The 4th International Workshop on Object-Oriented Reengineering*, 2003.

[7] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

[8] R. Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In *Proceedings of TOOLS39*, pages 103–116. IEEE Computer Society, 2001.

[9] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2002.

[10] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of ICSM04*, pages 350–359. IEEE Computer Society, 2004.

[11] R. Marinescu and D. Raţiu. Detection Of Design Problems in an Industrial Environment. In *Proceedings of SYNASC-2003*, 2003.

[12] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, third edition, 1996.

[13] P. Mihancea. Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems. Diploma thesis, "Politehnica" University of Timişoara, 2003.

[14] D. Parnas. Software Aging. In *Proceedings of The 16th International Conference on Software Engineering (ICSE94)*, pages 279–287. IEEE Computer Society Press, 1994.

[15] D. Raţiu, S. D. D. Gîrba, and R. Marinescu. Using History Information to Improve Design Flaws Detection. In *Proceedings of CSMR04*, pages 223–232. IEEE Computer Society, 2004.

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

[17] S. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach*. Prentice Hall, 1995.

[18] H. Sahraoui and M. Boukadoum. Extending Software Quality Predictive Models Using Domain Knowledge. In *Proceedings of The 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2001.

[19] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[20] M. Shepperd. Early Life-cycle Metrics and Software Quality Models. *Information and Software Technology*, 32(4):311–316, May 1990.