Detection Strategies: Metrics-Based Rules for Detecting Design Flaws

Radu Marinescu
Department of Computer Science
"Politehnica" University of Timişoara
Bvd. V. Pârvan 2, 300223 Timişoara (Romania)
radum@cs.utt.ro

Abstract

*In order to support the maintenance of an object-oriented software system, the quality of its design must be evaluated using adequate quantification means. In spite of the current extensive use of metrics, if used in isolation metrics are oftentimes too fine grained to quantify comprehensively an investigated design aspect (*e.g.*, distribution of system's intelligence among classes). To help developers and maintainers detect and localize design problems in a system, we propose a novel mechanism – called* detection strategy *– for formulating metrics-based rules that capture deviations from good design principles and heuristics. Using detection strategies an engineer can directly localize classes or methods affected by a particular design flaw (*e.g.*,* God Class*), rather than having to infer the real design problem from a large set of abnormal metric values. We have defined such detection strategies for capturing around ten important flaws of object-oriented design found in the literature and validated the approach experimentally on multiple large-scale case-studies.*

Keywords: metrics, object-oriented design, design flaws, quality assurance, design heuristics

1. Introduction

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 [14]. Thus, the identification and detection of these design problems is essential for the evaluation and improvement of software quality.

The fact that the software industry is nowadays confronted with a big number of large-scale legacy systems written in an object-oriented language, which are monolithic, inflexible and hard to maintain shows that 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 [17], [26] that must be known and used.

On the other hand, as De Marco pointed out [9] in order to control the quality of a design we need proper quantification means. Back in 1990 Card already emphasized [2] that metrics should be employed for the evaluation of software design from a quality point of view. But what should we measure? In the aforementioned context of design rules, principles and heuristics the question could be rephrased as follows: Is it possible to express these "good-design" rules in a quantifiable manner?

2. Problem Statement

The previous experiences of applying object-oriented metrics for the assessment and improvement of design quality in software systems are encouraging [11], [21], [18] but, the usage of metrics as it is now raises a set of new problems, summarized below:

- Beyond the problems concerning metrics definitions, – which are often imprecise, confusing or incomplete [20] – are the issues related to the *interpretation* of measurement results. In most of the cases no interpretation model or very empirical ones are provided, so that the applicability and reliability of measurement results is drastically hampered.
- The interpretation of individual measurements is too fine-grained if metrics are used in isolation; this reduces the applicability and relevance of metrics usage. Thus, in most cases individual measurements do not provide relevant clues regarding the *cause* of a problem. In other words, a metric value may indicate an

anomaly in the code but it leaves the engineer mostly clueless concerning the ultimate cause of the anomaly.

In order to synthesize the main point of our criticism we will use a medical metaphor: interpretation models can offer an understanding of symptoms reflected in abnormal measurement results, but they cannot bring the understanding of the disease that caused the symptoms. The *bottom-up approach – i.e.*, to go from abnormal numbers to the recognition of design diseases – is impracticable because the symptoms captured by single metrics, even if perfectly interpreted, may occur in several diseases. The interpretation of individual metrics is too fine grained to point out the disease. This leaves us with a major gap between the things that we measure and the issues that have an important quality impact at the design level.

Outline. In order to overcome this limitation, through the rest of this paper we propose a high-level, goal-oriented investigation mechanism, namely *detection strategy*, that allows us to define a usable top-down investigation approach. Section 3 defines detection strategies, and presents its key mechanisms *i.e.*, data filtering and data compositions. In Section 4 we propose then a lightweight methodology for defining detection strategies and exemplify each step on defining the detection strategy for the *God Class* design flaw. This is followed by a brief description of the tool support (Section 5) that facilitates the automatic detection of design problems. The entire approach is experimentally validated in Section 6 from the points of view of applicability and accuracy. The paper ends by discussing several related works and drawing several conclusions.

3. Detection Strategy

The main goal of this approach is to provide engineers with a mechanism that will allow them to work with metrics on a *more abstract level*, which is conceptually much closer to the real intentions in using metrics. The mechanism defined for this purpose is called *detection strategy*:

Definition 3.1 (Detection Strategy) *A* detection strategy *is the quantifiable expression of a rule by which design fragments that are conforming to that rule can be detected in the source code.*

A *detection strategy* is therefore a generic mechanism for analyzing a source code model using metrics. Note that in the context of the previous definition, "*quantifiable expression of a rule*" means that the rule must be properly expressible using object-oriented design metrics.

The use of metrics in the detection strategies is based on the mechanisms of **filtering** and **composition**. In the following subsections we will describe the two mechanisms in more detail.

3.1. The Filtering Mechanism

The key issue in data filtering is to reduce the initial data set, so that only those values are retained that present a special characteristic. This is usually called *data reduction* [16]. The purpose for doing that is to detect those design fragments that have special properties captured by the metric. The limits (margins) of the subset are defined based on the type of data filter. In the context of measurement activities applied to software, we usually search for *extreme (abnormal) values* or values that are in a *particular range*. Therefore we identify two types of filters:

- *Marginal Filter* – a data filter in which one extremity (margin) of the result set is implicitly identified with the corresponding limit of the initial data set.
- *Interval Filter* – a data filter in which both the lower and the upper limit of the resulting subset are explicitly specified in the definition of the data set.

3.1.1. Marginal Filters Depending on how we specify the limit(s) of the resulting data set, marginal filters can be *semantical* or *statistical*.

1. **Semantical.** For these filters two parameters must be specified: a *threshold value*, indicating the marginal value that must be explicitly specified; and the *direction*, that specifies whether the threshold value is the upper or the lower limit of the filtered data set. This category of filters is called *semantical* because the selection of the parameters is based on the particular semantic of the metric, captured by the interpretation model for that metric.
2. **Statistical.** In contrast to the semantical filters, statistical ones do not need an explicit specification for a threshold, as it is determined directly from the initial data set by using statistical methods (*e.g.*, box-plots, standard deviation). However, the direction must still be specified. Statistical filters are built based on the assumption that all the measured entities of the system are structurally designed using the same style and therefore the measurement results are comparable.

In our work we "instantiated" a set of concrete data filters from the two previous categories. Based on their practical usage and interpretation relevance these concrete filters can be grouped as follows (see also Figure 1):

- **Absolute Semantical Filters**: `HigherThan` and `LowerThan`. These filtering mechanisms can be parameterized with a numerical value representing the threshold. We will use such data filters to express "sharp" design rules or heuristics, – *e.g.*, "a class should not be coupled with more than 6 other

Type of Data Filter	Limit Specifiers		Filter Example
Marginal	Semantical	*Relative*	▪ `TopValues(10)`, ▪ `BottomValues(5%)`
		Absolute	▪ `HigherThan(20)` ▪ `LowerThan(6)`
	Statistical		▪ `Box-Plot`
Type of Data Filter	**Specification**		**Filter Example**
Interval	Composition of two marginal filters, with semantical limit specifiers of opposite polarities		`Between(20,30) := HigherThan(20) ∧ LowerThan(30)`

Figure 1. Classification of Data Filters

classes. Note that the threshold is specified as a parameter for the filter, while the two possible directions are captured by the two concrete filters.

- **Relative Semantical Filter**: `TopValues` and `BottomValues`. These filters are delimiting the filtered data set by a parameter that specifies the *number* of entities to be retrieved, rather than specifying the maximum (or minimum) value allowed in the result set. Thus, the values in the result set will be *relative* to the original set of data. The used parameters may be *absolute* (*e.g.*, "keep the 20 entities with the highest values") or *percentile* (*e.g.*, " retrieve the 10% of all measured entities having the lowest values"). This kind of filters is useful in contexts where we would like to consider the highest (or lowest) values from a given data set, rather than indicating precise thresholds. They should be therefore used for the quantification of design heuristics that are rather fuzzy, not specifying precise thresholds (*e.g.*, "methods of high complexity should be split"). For such design rules concepts of "high" and "low" are relative to the initial data set.
- **Statistical**: `BoxPlots`. Box-plots are a statistical method by which the abnormal values in a data set can be detected [13] [4]. Data filters based on such statistical methods, which are of course not limited to box-plots, are useful in the quantification of rules that although "fuzzy", concern with *extreme values* in a data set (*e.g.*, "Avoid packages with an excessively high number of classes"). Again, we have to specify the direction of the outlier values based on the semantics of the design rule that is quantified.

3.1.2. Interval Filters At first sight, for an interval data filter we need to specify two threshold values. But, in the context of detection strategies, where in addition to the filtering mechanism we also have a composition mechanism, interval filters are in all cases reducible to a composition of two marginal filters of opposite directions. In Figure 1 a simple example illustrates how the interval filter `Between(20,30)` can be composed out of two (semantical) marginal filters *i.e.*, `HigherThan(20)` and `LowerThan(30)`.

3.2. Selecting Data Filters

In the following, we provide a set of rules which should guide the engineer in deciding which type of data filter should be applied on the results of a particular metric, while quantifying the design rules or heuristics.

Rule 1 Choose an *absolute semantical filter* when quantifying design rules that specify explicitly concrete threshold values.

Rule 2 Choose a *relative semantical filter* when the design rule is defined in terms of fuzzy marginal values, like *"the high/low values"* or *"the highest/lowest values"*.

Rule 3 For large systems, parameterize *relative semantical filters* using *percentile* values. On the other hand, use *absolute* parameters when applying a relative semantical filter to small-scale systems.

Rule 4 Choose a *statistical filter* for those cases where the design rules make reference to extremely high/low values, without specifying any precise threshold.

3.3. The Composition Mechanism

In contrast to simple metrics and their interpretation models, a detection strategy should be able to quantify entire design rules. As a consequence, in addition to the filtering mechanism that supports the interpretation of individual metric results, we need a second mechanism to support a correlated interpretation of *multiple result sets* - this is the *composition mechanism*. The composition mechanism is based on a set of operators that "glue" together different metrics in an "articulated" rule.

We defined three composition operators: `and`, `or` and `butnot`. These operators can be seen from two different viewpoints:

- the *Logical Viewpoint*. From the logical perspective, the three operators are the reflection of the "joint-points" of a design rule quantified by the detection strategy, in which the "operands" are in fact a description of design characteristics (symptoms). They allow easy reading and understanding of the detection strategy because through the composition operators the quantified rule becomes similar to the original, informal one. From this point of view, for example, the `and` operator suggests the co-existence of both the symptom described on the left side of it as well as the existence of the symptom presented on the right side.
- the *Set Viewpoint*. The set viewpoint helps us to understand how the final result of a detection strategy is built (Figure 2). Through the filtering mechanism the initial set of measurement results for each metric involved in the strategy ($M_1 \dots M_n$) is reduced to a set of design entities (and their measured values) that were considered interesting in conformity with the interpretation model ($F_1 \dots F_n$). After that, the several filtered sets must be combined using the composition operators. Thus, in terms of set operators, the `and` operator will be mapped to *intersection ("$\bigcap$")*, the `or` operator to *reunion("$\bigcup$")* and the `butnot` operator to *set minus ("$\setminus$")*.

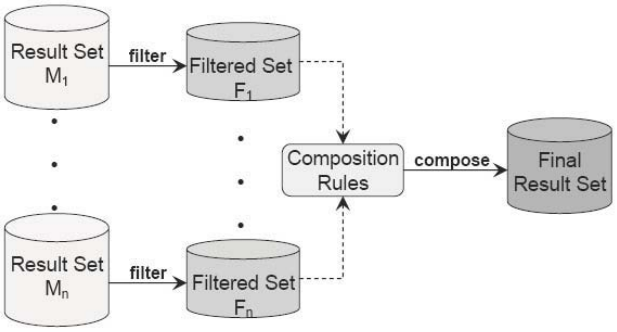

Figure 2. The filtering and composition mechanisms from a set viewpoint

3.4. The Issue of Threshold Values

Before closing this section, we would like to point out to an aspect that has a decisive influence on the accuracy of a detection strategy: the threshold values used in parameterizing any detection strategy. The problem is far from being new and it characterizes intrinsically any metrics-based approach. So, what are the "correct" threshold values to be used and how can we find them? In Section 3.1 we presented some hints towards a proper selection of filtering mechanisms, but we would like here to summarize and enlarge the perspective of that discussion. We identified three means of improvement on this issues:

1. *Experience and Hints from Literature.* In most of the cases setting the threshold values is highly empirical process and it is guided by similar past experiences and by hints from metrics' authors [19]. Where no hints are available we adapt threshold to the size of the system and set the thresholds rather *permissive*, as it is preferable to get more false positive results, rather than losing a real flaw due to a very strict threshold value.
2. *Tuning Machine.* A promising approach is found in [25] where the author defines a "tuning machine" which tries to find automatically the proper threshold values, and thus tune the detection strategies. This approach is based on building a repository of "flaw samples" *i.e.*, design fragments that have been identified by engineers as being affected by a particular design problem. Based on this reference samples, the "tuning machine" selects those threshold values maximize the number of correctly detected samples. The drawback of the approach is that the examples repository must be large enough to allow a proper tuning, while collecting these samples is not easy.
3. *Analysis of Multiple Versions.* Another relevant approach to this issue is to enhance the detection process by combining detection strategies applied on a single version with additional information about the history of the system [10]. Version analysis can provide for example information about the change stability of a class or the persistency of a design flaw over time. This approach is not intended to help parameterizing a strategy with thresholds, but it improves the accuracy of the detection process by adding for each suspect entity a valuable time perspective, which acts like an additional filter for those suspects.

4. Defining a Detection Strategy

In this section we will describe, based on the concrete example of *Behavioral God Classes* [26] the process of formulating detection strategies for a concrete design flaw. The starting point is given by one (or more) *informal rules* that define comprehensively the design problem. In this case we

will start from a set of three heuristics found in Riel's book [26]:

> *Top-level classes in a design should share work uniformly.* [...]
> *Beware of classes with much non-communicative behavior.* [...]
> *Beware of classes that access directly data from other classes.*

4.1. Stepwise Methodology

The first step in constructing a detection strategy is to break-down the informal rules in a correlated set of *symptoms* (*e.g.*, class inflation, excessive method complexity, high coupling) that can be captured by a single metric. In our case the first rule refers to a uniform distribution of intelligence among classes, and thus it refers to *high class complexity*. The second rule speaks about the level of intra-class communication between; thus it refers to the *low cohesion of classes*. The third heuristic addresses a special type of coupling *i.e.*, the direct access to instance variables defined in other classes. In this case the symptom is *access of "foreign" data*

The second step is to *select proper metrics* that quantify best each of the identified properties. For the *God Class* design flaw these properties are class complexity, class cohesion and access of foreign data. Therefore, we chose the following set of metrics:

- *Weighted Method Count* (WMC) is the sum of the statical complexity of all methods in a class [5]. We considered the McCabe's cyclomatic complexity as a complexity measure [23].
- *Tight Class Cohesion* (TCC) is the relative number of directly connected methods [1].
- *Access to Foreign Data* (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods [21].

The next step is to find the adequate filtering mechanism that should be used with each metric. This step is mainly accomplished based on the rules defined in Section 3.1 and 3.2. Thus, because the first symptom is a "*high* class complexity" we will choose the `TopValues` relative semantical filter for the WMC metric. For the "*low* cohesion" symptom we will choose again a relative semantical one, but this time obviously the `BottomValues` one. For the third symptom an absolute filter is needed as we want to capture *any* access to a foreign data; thus, we will use the `HigherThan` filter.

One of the most critical tasks in defining a detection strategy is to set the parameters (*i.e.*, the thresholds) for each data filter. In Section 3.4 we will discuss the different approaches to this issue, but for now we will use a simplistic approach and consider a 25% value for both the `TopValues` filter attached to the WMC metric and to the `BottomValues` filter of the TCC metric, meaning that we keep only one quarter of the classes in the system for each of the two metrics. Concerning the filter threshold for the ATFD metric, the decision is much simpler: no direct access to the data of other classes should be permitted so the threshold value is 1.

The final step is to correlate these symptoms, using the composition operators described in 3.3. From the context of the informal heuristics as presented by their author in [26], we infer that all these three symptoms should coexist if a class is to be considered a behavioral God Class. Therefore we used the `and`($\wedge$) operator to connect all symptoms. The final form of the detection strategy for God-Classes is formally described in Equation 1.

Alternatively, we have considered in the past [21] a "looser" formula for God-Classes which emphasized the relevance of the "acccess to foreign data" symptom over the " excessive complexity" and "lack of cohesion", as shown in Equation 2. Using one or the other variant strongly depends on the emphasis you want to set in a particular analysis.

Remark. The main of this paper is to use detection strategies to express rules that will help us detect *design problems* in object-oriented software system *i.e.*, to find those design fragments that are affected by a particular design flaw. At this point we want to emphasize that the *detection strategy* mechanism and the whole technique is not limited to problem detection, but it can serve other purposes as well. For example different investigation goals could be reverse engineering [3], detection of design patterns [21], identification of components in legacy systems [28], etc.

4.2. Implementation of Detection Strategies

To automatize the execution of detection strategies we need to express them in a computer understandable format. Therefore, we defined a simple interpreted language called SOD [6]. The language deals only with the filtering and composition of metrics results, relying on SQL queries for the computation of individual metrics (see Figure 4).

Remarks Beyond the technical details, the fact that the implementation of a detection strategy is all included in a single file helps us emphasize the achievement of the main goals of our approach:

1. A detection strategy is intended to *encapsulate* a mechanism for detecting a particular design flaw; in other words, everything needed for the detection of that design problem is hidden in a particular SOD file. One can use a detection strategy without knowing all the metrics and their correlations involved in the strategy.

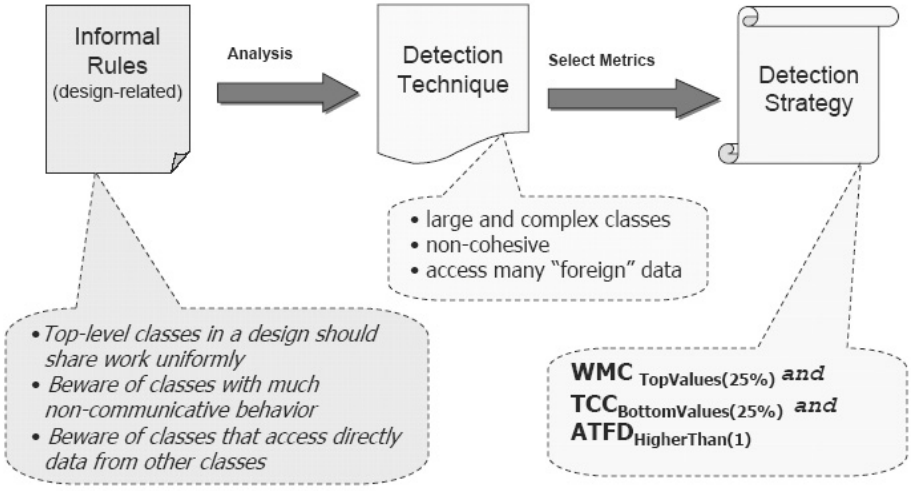

Figure 3. Process of transforming an informal design rule in a detection strategy.

$$GodClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (WMC(C), TopValues(25\%)) \wedge (ATFD(C), HigherThan(1)) \wedge (TCC, BottomValues(25\%)) \end{array} \right. \quad (1)$$

$$GodClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (ATFD(C), HigherThan(1)) \wedge ((WMC(C), TopValues(25\%)) \vee (TCC, BottomValues(25\%))) \end{array} \right. \quad (2)$$

Thus, the engineer can reason in the more abstract terms of detection strategies, rather than numbers.

2. Like a file, each detection strategy must have a suggestive *name*. Similar to the philosophy of patterns [15], the name of a detection strategy provides the strategy with an identity, which increases its communication level. Thus, name becomes an important vehicle of semantical knowledge. In order to be useful, names must capture the particular aspect quantified and detected by the strategy. This way a detection strategy helps the engineer not only to find critical design fragments, but also allows immediate reasoning about design improvements.

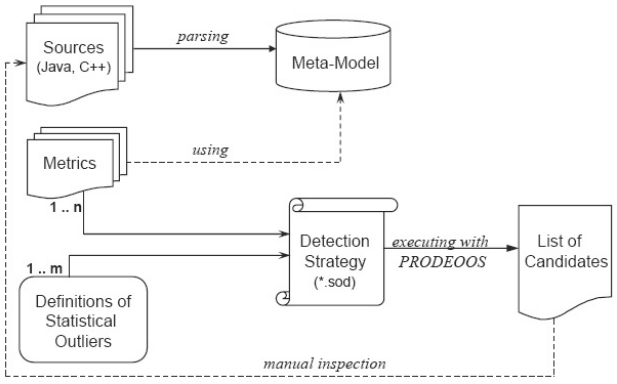

Figure 4. Overview of the inspection process.

5. Tool Support

Detection strategies would be probably useless in practice without an adequate toolkit that supports an automatic detection of design problems. Therefore, we developed the PRODETECTION toolkit that support code inspections based on detection strategies. The inspection process, summarized in Figure 4, consists of the following parts:

1. **Construction of the System Model.** In order to apply metrics on a given software system, we need to have all necessary design information. This is stored in a *meta-model* which consists of information about the design entities (*e.g.*, classes, methods, variables) of the system and about the existing relations (*e.g.*, inheritance, call

of methods) among these entities. This model is used as an abstraction layer towards the various programming languages that support object-orientation. Concretely, after the source code is parsed the extracted design model is stored in a relational database.

2. **Running the Detection Strategy.** A detection strategy implemented as a SOD script, can be automatically run using the PRODEOOS tool on the design model of the system to be analyzed. The result is a set of design entities together (*e.g.*, classes, methods) that are reported at suspect for that particular detection strategies. PRODEOOS returns not only the ultimate results, but also the values for the different metrics that were involved in the detection strategy.
3. **Verifying the Results.** The results obtained in the previous step must be manually inspected in order to decide if the suspects conform to the rule quantified in the detection strategy.

6. Evaluation

The evaluation of the approach was focused on the following desirable characteristics of a detection strategy used for the identification of design flaws:

1. Detection strategies should be *proper forms of expression* for a large variety of relevant design problems. In other words, can we indeed define such strategies for key design issues?
2. Detection strategies should properly *identify the problems*. Thus, a strategy is good if based on the symptoms (*i.e.*, the measurement results) it can detect the disease (*i.e.*, the design flaw) it claims to detect.
3. Detection strategies should be able *to localize the problem*. Therefore, a strategy is good if it able to identify those design fragments that are affected by a particular design flaw.

These three criteria fall in two categories: *applicability* and *accuracy* of the approach. The next two subsections address these two issues.

6.1. Evaluation of Applicability

The goal of our work is to formulate quantifiable rules that would help detecting design problems. In this context, in order to evaluate the applicability of the approach we have analyzed several important literature sources looking for design rules [24] [22] or descriptions of design flaws [14] [26] that have high impact on the quality of object-oriented design.

In defining the detection strategies we set up a twofold objective:

- ability to quantify design problems at three different levels of abstraction *i.e.*, at the method, class and subsystem level;
- ability to address problems related to four key criteria of good object-oriented design [8]: low coupling, high cohesion, manageable complexity and proper data abstraction.

As a result we succeeded in defining strategies for more than ten design problems, of which nine are mentioned in this paper. Addressing the issues of *improper coupling* detection strategies for *Shotgun Surgery*[14] (class level) and the *Wide Subsystem Interface*(subsystem level)[22] were defined. The issue of *cohesion* was quantified using two strategies: *Feature Envy*[14] (method level) and *Misplaced Class*[22] (subsystem level). Improper *distribution of complexity* can show up at all levels so we measured it using three strategies: *God Method*[14], *God Class*[26] and *God Package*[22]. Finally, the flaws related to *data abstraction* are captured using two class-level strategies *i.e.*, *Data Class*[26] and *Refused Bequest*[14].

6.2. Evaluation of Accuracy

Detection Strategies have already been applied successfully in the past on multiple large industrial case-studies, in the size of 700 KLOC up to 2000 KLOC especially on software for telecommunication. To assess the accuracy of the suite of detection strategies mentioned in the previous section (6.1) we used two successive versions of a medium size business application relate to computer-aided route planning. We also knew that the second one is a re-engineered and enhanced version of the first system[1]. The size characteristics of the two systems is summarized in Table 1.

System	KLOC	Packages	Classes	Methods
SV1	93	18	152	1284
SV2	115.6	29	387	3446

Table 1. Size of the two case-study versions

For evaluating the accuracy of detection strategies we defined a twofold approach :

Automatic Classification. The main idea is to split the suspects (indicated by a detection strategy) in the first version of a system (SV1) in two categories: real flaws and false positives. We differentiate between the two categories based on the following assumption: a suspect is affected by a *false*

1 Through the rest of this section, we will designate the two versions of the system as SV1 and SV2, standing for *System Version 1* and *System Version 2*.

positive if it reoccurs in the list of suspects in SV2, for the same detection strategy; otherwise a *real flaw* was detected. By dividing the number of real flaws by the total number of suspects we get the *accuracy rate* for the given detection strategy. The main advantages of this approach are that it is highly *objective* and it can be easily *automated.*

Manual Investigation. Because the *Automatic Classification* approach is "blind" based on a reasonable yet fallible assumption, it doesn't give us the certainty that the classification of the suspects, and consequently the computed accuracy factor are correct. Therefore, we decided to use in addition to the described approach a further manual investigation, which involves the manual inspection of all the suspects. The manual investigation allows a proper evaluation of both accuracy aspects that are relevant for a detection strategy *i.e.*, it allows us to identify those suspect design fragments, which although ill-designed, are not affected by the design-flaw quantified by the detection strategy (see the criteria defined in the beginning of Section 6).

6.3. Discussion of Results

Figure 5 summarizes the results of applying a suite of detection strategies to the two versions of the analyzed system. The table also contains the partitioning of the suspects in SV1 in conformity with the rules of the *Automatic Classification* approach In the last column of the table we see the *accuracy rate* for each evaluated detection strategy. The accuracy rate is computed as the number of real flaws divided by the total number of suspects in SV1 [2] .

Design Flaw	Suspects in SV1	Suspects in SV2	False Positives	Special Cases	Real Flaws	Accuracy Rate
Feature Envy	40	15	11	4	25	63%
God Method	4	4	1	3	3	75%
Shotgun Surgery	15	7	6	1	9	60%
Refused Bequest	22	6	4	2	18	81%
God Class	5	2	2	0	3	60%
Data Class	3	2	1	1	2	66%
God Package	2	1	1	0	1	50%
Misplaced Class	4	2	1	1	3	75%
Wide Subsys. Interface	5	1	1	0	4	80%

Figure 5. Accuracy scores based on the *Automatic Classification* approach

In Figure 6 we have summarized the results of the *Manual Investigation* approach for a subset of the previously analyzed strategies. In this case, we only analyzed the suspects from the first version of the system (SV1). The total number of suspects for a detection strategy is classified in three categories, after a manual inspection: *Correct Detection* contains those suspects that have proved to be affected by the design flaws that the detection strategy claims to find; the *Other Flaw* category contains the suspects that have proved to be ill-designed but are affected by another design problem than the one supposed to be captured by the strategy; the last category, *False Positive* includes all those suspects that revealed no design problems during the manual inspection. Using the manual investigation approach we

Design Flaw	Total Suspects	Correct Detection	Other Flaw	False Positive	Strict Accuracy	Loose Accuracy
God Method	4	2	1	1	50%	75%
Shotgun Surgery	15	10	2	3	66%	80%
God Class	5	3	1	0	80%	100%
Data Class	3	3	0	0	100%	100%
Wide Subsys. Interface	5	3	1	1	60%	80%

Figure 6. Accuracy scores for SV1 using the *Manual Investigation* approach

can compute two types of accuracy rates: a *strict* one that only counts the cases in which the "diagnosis" was correct and *looser* one that counts all the suspects that were indeed ill-designed.

We distinguished between two aspects of the accuracy criterion: one that evaluates the ability of a detection strategy to find ill-designed code fragments (loose accuracy) and another one that deals with the ability of a strategy to find only those design entities that are affected by the particular flaws that the strategy claims to detect (strict accuracy). Using the *Automatic Classification* approach we could only measure the strict accuracy of the evaluated strategies. For all the strategies the accuracy rate is over 50%, while the average accuracy is over 67%. The strategy with the lowest accuracy rates (50%) is: *God Package*, which we believe is due to the low absolute number of suspects (2).

Using the *Manual Investigation* evaluation method we could compute both the *loose* and the *strict* accuracy rate. Because the manual investigation is a time consuming operation we limited it to only a subset of detection strategies. The average *loose accuracy rate* was in this case 87%, substantially higher than the one obtained using the automatic approach. At this point we also need to mention that almost all suspects that were classified as *Other Flaw* (see Figure 6) were affected by a flaw which proved to be correlated to the one quantified by that particular detection strategy. Thus, a first conclusion is that the strategies that we defined so far have a high accuracy in detecting ill-designed

2 In the *Special Case* column we count those entities which were either renamed or moved to another place in the system. Because we were not totally if that entity is corresponding to an entity from SV1 we decided to count them separately.

entities.

As we compute the average of the *strict accuracy rate* the result is around 70%. Thus, in more than two out of three cases the suspect entity has been indeed affected by the expected design flaw, which we consider to be an acceptable accuracy rate.

7. Related Work

This work is at least related with contributions that fall in the following two research areas: first, those focused on the *quantification of design rules and the detection of bad smells*; second, those concerned with the *use of correlated metrics for design inspections*.

Quantification of Design Principles and Rules. As discussed earlier, most design rules and heuristics are defined informally in the literature. Yet, sometimes even among rules defined by the same author we find some which are directly quantifiable (*e.g.*, Riel states that "a class should not contain more than *six* objects" [26]). In this context, a remarkable contribution comes from Martin [22] who beyond formulating almost a dozen of key principles, proposed a quantification for two of the principles of subsystem design *i.e.*, the principles of "Stable Dependency" and "Stable Abstraction". For this, he defines two metrics together with a precise interpretation of the two principles in terms of these metrics. The limitation of the aforementioned contributions is that they are based only on a direct mapping between a principle or rule and one or two metrics, which is hardly possible for most design rules. Concerning the detection of design rules' violations, Ciupke [7] proposes an approach in which the rules are specified in terms of queries, usually implemented as Prolog clauses. The approach is applicable for rules that can be specified in a "sharp" manner (see Section 3.1) but is hardly usable for rules defined in more general terms. Recently, an automatic approach of "code smells" detection was proposed in [29], but it is quite much focused on implementation issues (*e.g.*, code conventions, type casts) rather than design problems.

Using Correlations of Metrics for Design Inspections. The idea to use metrics in combination to understand the design of a piece of software and/or assess its quality is also encountered in several approaches. Erni [12] introduces the concept of multi-metrics, as an n-tuple of metrics expressing a quality criterion (*e.g.*, modularity). Unfortunately, multi-metrics do neither encapsulate metrics in a more abstract construct, nor do they allow a flexible combination of metrics. In a further approach named "class blueprint" [18], multiple metrics are used in the context of visualization techniques for the understanding of classes' structure. Here, the elements of classes are visualized as rectangular boxes, whereby the dimensions and color of these boxes are defined by a set of metrics. Obviously, the approach can be used also for detecting suspicious "blueprints" of classes. Yet, compared to detection strategies, the use of metrics in the aforementioned approach has to limitations: on one hand, it lacks a proper filtering mechanism as in the "class blueprint" filtering is mainly done visually; on the other hand, only few metrics (2-3) can be used simultaneously and furthermore these can be combined only in a very restricted manner. In a recent contribution, [27] the authors use a specially selected set of seven metrics to detect portions of code that need to be refactored. The metrics are used within a "diagnosis algorithm", which puts in correspondence different combinations of metric values with a desirable code refactoring. Yet, the approach does neither provide any mechanism for combining explicitly metrics in a quantifiable rule, nor does it offer a direct link to design rules and principles.

8. Conclusions. Future Work

We have presented detection strategies as a novel mechanism for formulating metrics-based rules that capture deviations from good design principles and heuristics. The main features of our approach are:

- It offers a *higher abstraction level in working with metrics*, because a detection strategy can directly – and automatically, if supported by tools – localize design fragments affected by a particular design flaw rather than having to infer the real design problem from a large set of abnormal metric values.
- It provides a proper means to *express relevant "good-design" rules and heuristics in a quantifiable manner*. In Section 4 we have shown that the process of defining detection strategies is easy to follow, making it usable for any software engineer or maintainer dealing with the design quality of an object-oriented system.

We have validated the applicability of the approach by defining more than ten detection strategies that capture various design problems. We have also shown based on an industrial case-study that strategies are accurate means of problem detection, and are usable in practice. It is clear that the reduced number of case studies is by no means statistically relevant, but nevertheless it is cleat that the approach warrants further study and experimentation.

We will focus our future work on three main fronts:

- We will address in more detail the issue of choosing threshold values for a detection strategy. Starting from the remarks made in this paper we want to build a coherent and applicable framework for selecting thresholds, increasing thus the accuracy of detection.
- We intend to continue the validation against other software systems and experimenting with different metrics that could be used alternatively in a detection strategy.

- As there are also design problems that cannot by currently expressed using detection strategies, we intend to research the possibility of extending the mechanism so that a unitary approach to problem detection can be established.

Acknowledgments This work is supported by the Austrian Ministry BMBWK under Project No. GZ 45.527/1-VI/B/7a/02.

I would like very much to thank Tudor Gîrba, Daniel Raţiu and and Cristina Marinescu for all their encouragement, the fruitful discussions and for reviewing this paper. I would also like to thank Ciprian Chirilă and Petru Mihancea for helping me with a lot of the implementation effort related to my research. Last but not least I would like to thank the LOOSE Research Group (LRG) for being such a great team.

References

[1] J.M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. *Proc. ACM Symposium on Software Reusability*, apr 1995.

[2] D. Card and R. Glass. *Measure Software Design Quality*. Prentice-Hall, NJ, 1990.

[3] E. Casais. State-of-the-art in Re-engineering Methods. achievement report SOAMET-A1.3.1, FAMOOS, October 1996.

[4] J. Chambers, W. Cleveland, B. Kleiner, and P. Tukey. *Graphical Methods for Data Analysis*. Wadsworth, 1983.

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

[6] C. Chirilă. Instrument Software pentru Detecţia Carenţelor de Proiectare în Sisteme Orientate-Obiect. Diploma Thesis, "Politehnica" University Timişoara, 2001.

[7] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Technology of Object-Oriented Languages and Systems - TOOLS 30, IEEE Computer Press*, 1999.

[8] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, London, 2 edition, 1991.

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

[10] D.Ratiu, S. Ducasse, T. Girba, and R. Marinescu. Using History Information to Improve Design Flaws Detection. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*. IEEE Computer Society, 2004.

[11] K. Erni. *Anwendung multipler Metriken bei der Entwicklung objektorientierter Frameworks*. PhD thesis, Universität Karlsruhe, Mnster, 1996.

[12] K. Erni and C. Lewerentz. Applying Design-Metrics to Object-Oriented Frameworks. In *Proceedings 3rd International Software Metrics Symposium*, pages 64–74, Los Alamitos, 1996. IEEE Computer Science Press.

[13] N. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

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

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] P.G. Hoel. *Introduction to Mathematical Statistics*. Wiley, 1954.

[17] R.E. Johnson and B. Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[18] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *OOPSLA 2001 proceedings*, 2001.

[19] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall Object-Oriented Series, Englewood Cliffs, NY, 1994.

[20] R. Marinescu. The Use of Software Metrics in the Design of Object-Oriented Systems. Diploma Thesis, "Politehnica" University Timişoara, 1997.

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

[22] R.C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall: 1st Edition, 2002.

[23] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, dec 1976.

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

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

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

[27] L. Tahvildari and K. Kontogiannis. A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations. In *CSMR 2003 proceedings*, 2003.

[28] A. Trifu. Using Cluster Analysis in the Architecture Recovery of OO Legacy Systems. Diploma Thesis, Karlsruhe and the "Politehnica" University Timişoara, 2001.

[29] E. van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *WCRE 2002 proceedings*, 2002.