Detecting Design Flaws via Metrics in Object-Oriented Systems

Radu Marinescu
"Politehnica" University of Timişoara
Department of Computer Science
Bvd. V. Pârvan 2, 1900 Timişoara, Romania
radum@cs.utt.ro

Abstract

The industry is nowadays confronted with large-scale monolithic and inflexible object-oriented software. Because of their high business value, these legacy systems must be reengineered. One of the important issues in reengineering is the detection and location of the design flaws, which prevent an efficient maintenance and further development of the system. In this paper we present a metrics-based approach for detecting design problems, and we describe two concrete techniques for the detection of two well-known design flaws found in the literature. We apply our technique on an industrial case-study and discuss the findings. The experiment shows that the proposed technique found indeed real flaws in the system and it suggests that, based on the same approach, further detection techniques for other common design-flaws can be defined.

Keywords: Reengineering, metrics, design flaws, problem detection

1 Introduction

In the beginning of object-orientation it was hoped that the use of object-oriented mechanisms like inheritance, encapsulation or polymorphism, will make the software systems more flexible, more extensible and understandable, and thus they will become easier to maintain. Nowadays, the industry is confronted with a large number of software systems, consisting of millions of lines of code written in an object-oriented language, that proved after years to have few of those expected qualities; instead of that they are monolithic, inflexible and hard to extend. In order to keep over time the value of these systems, – both technical and economical – they must be *reengineered* [9].

Reengineering is a difficult and complex process that requires a high amount of effort. Big part of the effort is spent on identifying those parts of the system that are affected by particular design-flaws, and which need to be redesigned in order to achieve the reengineering goal. This task is called *problem detection* [2]. Except for some recent efforts [3], problem detection is currently approached in a highly parochial and mostly non-automatic manner which makes it:

- *time expensive,* because its lack of rigor in detecting design flaws makes it very hard to use in practice, requiring therefore a lot of time and energy;
- *unrepeatable,* because problem detection is done on a case-to-case basis and is therefore limited towards its further employment in a different context;
- *non-scalable,* because a non-automatic detection method is impossible to be applied on large-scale systems. Unfortunately most industrial systems have millions of lines of code, and therefore an automatic approach to problem detection becomes a necessity.

0-7695-1251-8/01 $10.00 © 2001 IEEE

Consequently, problem detection needs a *systematic*, *repeatable* and *scalable* approach in order to increase its efficiency and applicability. In this paper we describe a metrics-based approach for problem detection that fulfils the criteria mentioned before, and we examine how this approach is concretely used for the detection of two well-known design-flaws defined in the literature, i.e. *"god-classes"* [12] and *"data-classes"* [7]. *God classes* are those classes that tend to centralize the intelligence of a system, while *data-classes* are those that define data fields and almost no methods except some accessor methods. We conducted an experiment on an industrial case-study, in which we applied the detection techniques. These first results indicate that our approach can be successfully used for problem detection. They also make show that using the general method defined by this approach, similar techniques can be defined for the detection of other design-flaws.

This paper is structured as follows. Section 2 presents the approach and defines a consistent format for the description of particular detection techniques. Next, based on the described approach, and using the defined format, we introduce in Section 3 concrete techniques for the detection of two well-known design flaws and discuss the results of applying them on the case-study. In Section 4 we describe the tools that were used for the implementation of the approach. Section 5 discusses the related work, comparing it in each case with our approach. Section 6 summarises the contribution of the paper and points to the main issues that we intend to address in the future.

2 A Metrics-Based Approach for Problem Detection

2.1 The Approach

The starting point for this approach is an informal description of a design-flaw. In the recent years we found often and in various forms in the literature descriptions of flawed design structures. These are what Fowler calls "bad-smells" [7]. Riel presents in [12] a set of heuristical design guidelines, and discusses some of the flawed structures that result if these guidelines are violated. In the same manner, Martin discusses the main design principles of object-orientation and shows that their violation leads to a "rotting design" [10].

The approach consists of the following sequence of steps:

1. *Quantitative Analysis of the Design-Flaw.* After choosing the design-flaw that should be detected, the first step is to express the informal description of the flaw in a quantitative manner. In other words we have to describe how the flaw affects the design entities (e.g. class inflation, excessive method length, etc.) and the relationships among them (e.g. highly coupled subsystems). The results of this step is the definition of a concrete detection strategy, based on the analysis of the informal description.
2. *Selection of Metrics.* Based on the quantitative description of the design problem, those metrics must be found or defined, that are most suitable to measure the characteristics of the flawed design structure. At the end of this step, the detection strategy can be expressed as a specific combination of the selected metrics.
3. *Detection of Suspects.* The third step is to measure the system based on the defined detection strategy, using the chosen metrics. The result of this step will be a list of design fragments that are suspect of that design-flaw.
4. *Examination of Suspects.* The last step is to examine the suspect design fragments and to decide, based on the source-code and on other information sources, if the suspects are indeed flawed or if there is a conscious design decision behind that design-fragment.

Concerning these four steps, there is an important distinction to be made between them: while the first two steps describe the *definition* of the detection technique, the last two describe the *application* of the technique on the examined system. Thus, after a detection technique is defined and validated, only the last two steps must be covered for detecting a particular flaw in a given system.

2.2 Template for Describing a Detection Technique

We describe each particular detection technique using a consistent format. This format lends a uniform structure to the detection process, making it easier to understand and apply. The *definition* of a detection technique is divided in sections, according to the following template:

- *Motivation.* In this section the design-flaw is shortly explained with a special focus on its impact on the external quality attributes of the design (e.g. flexibility, maintainability, etc). From this description we derive the characteristics of the structural design entities (e.g. classes, methods, subsystems) that are affected by that design problem.
- *Strategy.* This section describes the concrete strategy to use in order to find the flawed entities. The strategy is based on the characteristics of the flawed design-entities that were presented in the *Motivation* section.
- *Metrics.* As our approach is a metrics-based one, we need to select those metrics that we are going to use for the detection of that particular design-flaw. This section introduces the metrics that are needed to support the detection strategy described before. For each metric three aspects must be provided: the *definition* of the metric, its *interpretation model* and the specification of *outliers*. Note that a metric might be used in several detection techniques, and while the definition of the metric is independent of the context where it is used, its interpretation model and sometimes the outliers are related to the particular technique.

In addition to these three sections that organize the definition of the technique, the next two sections are structuring the way the technique is *applied* to a given system:

- *Measurements.* This section contains the results of measuring the system and some overall remarks concerning the measurement results. The results must point out to the suspect design fragments.
- *Findings.* This section discusses the findings of the detection technique, i.e. the results of examining the suspect entities found by measurements. As we will see later in this paper (Section 3.4), it is is important to describe not only the real findings, but also the false positives, as they might describe a recurring context where the technique does not apply.

3 Detection of Design Flaws

In this section we use the approach presented before to detect two design-flaws. The technique is applied to an industrial software system, that we briefly introduce in the beginning of this section. The section is concluded by discussing the applicability and the limitations of the approach.

3.1 The Case-Study

The analysed system is a medium-size business application[1] written in C++. The application is composed of 12 subsystems hierarchically organised. The other size characteristics are summarised in the table below:

LOC	Classes	Methods	Free Functions
50000	91	581	158

Table 1. A summary of the case study

[1] Since the case study is subject to a non disclosure agreement we cannot offer more information about it.

The application was developed in the recent years (1997-1998) by designers and programmers familiar to object-orientation. One year later the developers requested a reengineering of the system, which proves that a number of desing weaknesses were encountered. This justifies our search for design flaws in this case-study.

3.2 Data-Classes

3.2.1 Motivation

Data-classes are dumb data holders and almost certainly other classes are strongly relying on them. The lack of "functional" methods may indicate that related data and behaviour are not kept in one place, this is a sign of a non object-oriented conception. In terms of quality attributes, such classes affect:

- *maintainability*, because changes in the data-class will almost sure affect the other classes that use its data;
- *testability*, because if two or more classes use some of the data in exactly the same way, duplicated code will appear and consequently the effort needed to test this code will increase, compared to testing just one method placed in the data-class.
- *understandability* of the classes that use the data-classes, because of the functionality that semantically does not belong to that class, but to the "data class".

3.2.2 Strategy

We will detect data-classes based on their characteristics: we search for "lightweight" classes, i.e. classes which provides almost no functionality through its interface. Next, we will look for the classes that define many *accessor methods* (get/set methods) and for those who declare data fields in their interface. Finally, we will confront the lists and manually inspect the "lightweight" classes that declare many public attributes and those who do provide many accessor methods.

3.2.3 Metrics

1. **Weight of a Class (WOC)**
   - *Definition:* WOC is the number of non-accessor methods in a class divided by the total number of members of the interface. Inherited members are not counted.
   - *Interpretation:* The lower the WOC value for a class, the more the class is suspect to be a data-class. Well designed classes tend to have a WOC value of 1.0, or very close to this value, because the interface of the class should contain only the methods that implement the behaviour of the class.
   - *Outliers:* Classes with values between 0 and 0.33. This interval represents the lower third of the whole range of possible values. Thus, the outliers are classes where more than 2/3 of the public members are not functional methods. We decided to select the outliers by this rule, in order to get only the extreme cases of "lightweight" classes.
2. **Number Of Public Attributes (NOPA)**
   - *Definition:* NOPA is defined as the number of non-inherited attributes that belong to the interface of a class.
   - *Interpretation:* Classes with public data members violate encapsulation and couples its clients to its structure.
   - *Outliers:* The top-ten classes, but not having NOPA values less than 3. We decided to use this composed rule, in order to get only those classes where the declaration of public attributes, is used extensively. Classes having a NOPA value less then 3, might be also data-class suspects, but are supposed to be smaller and there having less impact on the overall design.

3. **Number Of Accessor Methods (NOAM)**
   - *Definition:* NAM is defined as the number of the non-inherited accessor methods declared in the interface of a class.
   - *Interpretation:* If a class has a high NOAM value a part of the functionality of that class is probably misplaced in one or more other classes.
   - *Outliers:* The top-ten classes, but not having NOAM values less than 3. In establishing this outlier criteria, we used the same considerations as for the NOPA metric, described before.

3.2.4 Measurements

We decided that a class becomes a data-class suspect if it is a WOC outlier and in the same time it is among the outliers of at least one of NOPA and NOAM. The results are summarised in Table 2 which shows the outliers for the WOC metric and their results for the other two metrics, based on which the suspect classes were selected. A dash (—) indicates that the class is not among the outliers for that metric.

Class	WOC	NOPA	NOAM	Suspect?
StreetRecord	0.08	8	3	YES
TownRecord	0.21	12	3	YES
Segment	0.25	6	—	YES
CPredicate	0.25	—	—	NO
CScanner	0.33	—	—	NO
CAttributePredicate	0.33	—	—	NO
CRangePredicate	0.33	—	—	NO
StreetNameRecord	0.33	5	—	YES
PartSegments	0.33	—	—	NO

Table 2. The results for detecting "Data-Classes" in the case-study.

Thus, we selected 4 classes for further manual inspection : `StreetRecord`, `TownRecord`, `Segment` and `StreetNameRecord`. Analysing the rest of the classes we found that indeed all these classes are nothing more than dumb data holders, with as good as no functionality.

3.2.5 Findings

In order to evaluate the impact of these data-classes to the rest of the system, we took a look at those classes which *use* these data and on the semantic relation between the user-classes and the data-class. Here are the findings:

- *Relation of* `StreetRecord`, `StreetNameRecord` *and* `TownRecord` *to their clients*
  The data-classes are in fact data records used by classes that manage the persistence of the data kept in the record objects. These data-classes might be redesigned, but their impact on the other classes is small so that the overhead of a redesign is not justified.
- *Relation between* `Segment` *and the client-class* `SuperPolygonArray`
  Class `SuperPolygonArray` contains three object arrays that belong semantically together – one for `Segment` and two more for `SuperPolygon` and `MapPoint`, which are smaller data-classes. All these three classes are treated as dumb data holders and their data is manipulated directly by `SuperPolygonArray`

The *Data-Classes* design flaw is also referred by Riel[12], in connection with "god-classes", which is the design-flaw described next in this paper. Thus the two design-flaws that we discuss in this paper, although different, are related.

3.3 Behavioural God-Classes

3.3.1 Motivation

An instance of a god-class performs most of the work, delegating only minor details to set of trivial classes and using the data from other classes. The detection and redesign of the "god-classes" is motivated by the negative impact that these classes have on at least the following quality attributes:

- *reusability*, because they will incorporate more than one functionality, reducing the possibility of using them in a different context.
- *understandability*, because the responsibility of the class is no longer clear as it is "overloaded" with a lot of functionality that semantically belongs elsewhere.

3.3.2 Strategy

The detection of god-classes is based on the three characteristics these classes: they are expected to access a lot of data from "lightweight" classes (direct or through accessor methods), they are expected to be large and to have a lot of non-communicative behaviour. Like we did before, we will first detect the classes which strongly depend on the data of "lightweight" classes. After that, we will filter the first list of suspects, by eliminating all the small and cohesive classes.

3.3.3 Metrics

1. **Access Of Foreign Data (AOFD)**
   - *Definition:* AOFD represents the number of external classes from which a given class accesses attributes, directly or via accessor methods. Inner classes and superclasses are not counted.
   - *Interpretation:* The higher the AOFD value for a class, the higher the probability that the class is or is about to become a god-class.
   - *Outliers:* The top-ten classes, but not having AOFD values less than 3. Because god-classes are expecte to use *a lot* of data from other classes, we defined this outlier rule in order to avoid the classes that might use only few data from other classes, as we believe that such classes have a lower impact on the overall design.
2. **Weighted Method Count (WMC)** [4]
   - *Definition:* WMC is the sum of the statical complexity of all methods in a class. If this complexity is considered unitary, WMC measures in fact the number of methods (NOM).
   - *Interpretation:* WMC measure both sizes and complexity: the higher the WMC value for class the larger and more complex the class is. Consequently, the outliers are major abstractions in the system, but also possible god-classes.
   - *Outliers:* The top-ten classes in the system. We are interested only in the most complex classes from the system, and therfore we decided on this outlier rule. Smaller classes, although they might have the tendency to become gods, are not as relevant as the larger and more complex ones.
3. **Tight Class Cohesion (TCC)** [1]
   - *Definition*: TCC is defined as the relative number of directly connected methods. Two methods are directly connected if they access a common instance variable of the class.
   - *Interpretation:* The lower the TCC value for a class is, the more non-communicative behaviour in the class.
   - *Outliers:* Classes with values lower than 0.33. This criteria was defined in order to capture only those classes where less than one-third of the method-pairs in a class are connected via instance variables. This decision is based on our previous experience with this metric and on the indications found in the literature.

3.3.4 Measurements

The results are summarised in Table 3 which shows the outliers for the AOFD metric and their results for the other two metrics, based on which the suspect classes were selected. A dash (—) indicates that the class is not among the outlier for that metric.

Class	AOFD	AOF/WOC	WMC(NOM)	TCC	Suspect?
CParser	7	4	75(19)	0.28	YES
SuperPolygonArray	5	3	54(13)	—	NO
StreetNameTable	5	1	44(10)	—	NO
CTownNearestSearch	4	2	—	—	NO
CStreetSegmentRangeSearch	3	1	—	0.35	NO
TownView	3	1	43(11)	—	NO

Table 3. The results for detecting "God-Classes" in the case-study.

Based on these results and on our selection criteria, we choose for further inspection only one class: `CParser`. We inspected this class, by analysing its relations to the data-classes that it was using and the conclusion was that the `CParser` definitely centralises almost all the functionality and uses the other classes only to get or to set their data.

Although we didn't select the `SuperPolygonArray` class for inspection, it is no surprise to find it among the outliers of AOFD and WMC, if we consider the finding presented in the previous section. As we have seen there, this class should also be redesigned.

3.3.5 Findings

- **Relation between class `CParser` and the predicate classes (`CPredicate`)**
  Class `CParser` parses a string representing a selection query (SELECT) and keeps two lists: one containing the selected columns and one containing the predicates that compose the conditional clause (WHERE) of the selection. The predicates can be of different types, each one being parsed differently. In the current implementation, for each predicate type a class is defined. These classes are in fact some "stupid" records that contain only the fields that keep the specific data specific for each type of predicate. In the `CParser` class, for each predicate type a method is defined that parses the predicate and sets the fields in the corresponding predicate class.
- *Relation between classes* `CParser` *and* `CScanner`.
  Class `CScanner` is in fact a string tokenizer used by the `CParser` class to parse a SQL selection statement. The `CScanner` class keeps a field that indicates the type of the current token. If the token is a number or an identifier the scanner must also provide the value of the number or the string. In the current design `CScanner` is a structure that exposes all the implementation details, and `CParser` directly reads the fields from the tokenizer.

3.4 Discussion

After we have presented the results and the findings for the two detection techniques, two main issues need to be addressed: do these techniques apply to other case-studies? and: does the approach generalise to other design-flaws? We will next briefly tackle these two issues.

Concerning the first issue, although we fully agree that more case-studies are needed in order to ultimately validate the techniques, we consider the selected case study relevant because of two reasons: first, because it is an industrial case-study, that comes from a extremly dynamic area of the software development, which tends to alter the initial design concept of a system; second, because the reengineering of this system was required directly by the designers of the system themselves. On

the other hand, the results show us that there are typical situations (e.g. classes that manage the persistence of data, inner classes) where a detection technique will find wrong or irrelevant suspects. By analyzing more systems these recuring situation can be identified and documented, making the detection more focused.

The second issue refers to the extent at which the approach generalises for other design problems. This approach applies to all those design-flaws that can be expressed in form of metrics that rely on the source-code. We have already defined and are currently doing extensive experiments on a number of 6-7 detection strategies inspired by the "bad-smells" found in [7] (e.g. detection of "Shotgun Surgery", "Refused Bequest" or "Feature Envy"). Besides these, we identify two categories of other flaws that cannot be addressed by this approach: first, there are design-flaws that can hardly be expressed in form of measurement (e.g. *"duplicated code"*); second, there are cases where the flaw is described in terms of metrics that rely on other artifacts than the source-code (e.g. "Number of Detected Bugs per Class" relies on the available bug-reports). Yet, we estimate that more than 80% of the design-problems referenced in the literature are addressable by our approach.

4 The Tool Support

In order to reduce the time spent on problem detection we need tools that automatise the detection to a large extent. An overview of the tools and of their operation sequence is depicted in Figure 1. From the tools perspective, following steps are implied in the detection process. First, the source files are parsed and the design information is extracted. This step detaches the approach itself from the lower abstraction level of the concrete implementation and brings it to the higher level of abstraction offered by a *design model*. This model is used as an an abstraction layer towards the various programming languages that support object-orientation[2]. Second, the metrics used for the detection are computed. These metrics are implemented as SQL queries based on the tables containing the design model. Having the tables and the implementation of the metrics, the measurement results can be computed. In the end, the results are manually investigated based on the detection strategy.

TABLEGEN is a own-developed tool which extracts *design information* from the C++ source-code and stores it in form of plain-text tables. The tables contain information about all the classes, methods and variables defined in the analysed project, about the inheritance relations among classes and, last but not least, it delivers information about method calls and variable accesses. Compared to other similar tools it extracts substantially more information and is more stable.

5 Related Work

The idea to automatise the problem detection phase is not new, neither is the idea to use metrics in order to improve the quality of software systems. In this section we will therefore present a short summary of three recent contributions that are related to our work.

Problem Detection Based on Design Querying. In [3] we find an alternative approach to problem detection based on violations of design rules and guidelines. This approach focuses on structural properties that can be detected fully automatically, pointing directly to the "critical design fragments". These properties are specified as "sharp" rules, the result of applying such a query being a bipartite decomposition of the entities of the system, i.e. those which violate the rule and the others. Compared to this method, the metrics-based approach is focused more on "fuzzy" guidelines – e.g. "a class should not be strong coupled to other classes" – which require a *multi-partite decomposition* of the entities (e.g. "very strong coupled", "normally coupled", "loosely coupled").

[2] Although the experiment described in this paper is made on a C++ project, the approach is not language dependent. A similar tool that would extract the same design information from Java code is also available.

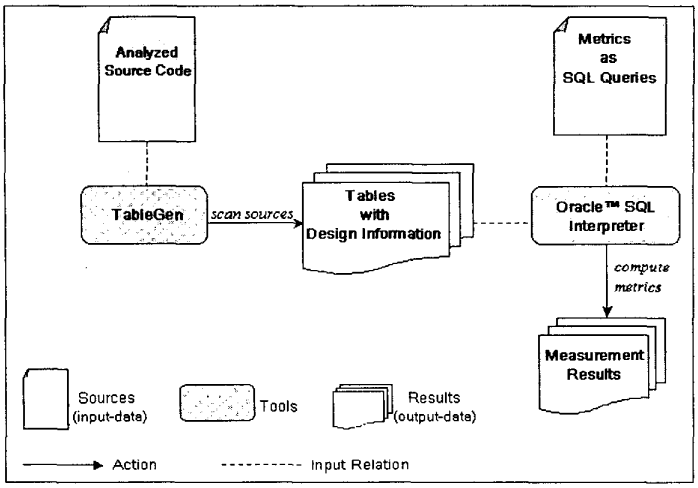

Figure 1. Overview of the tools and their operation sequence

Using Metrics for Improving Framework Development. In [6], Erni proposes a metrics-based approach for improving the consolidation phase of framework development, by speeding-up the detection of the "hot spots" in the framework. The approach is in some extent similar to problem detection and it offers therefore some interesting input for our approach – e.g. measuring the entities in their context, defining a quality model, or using metrics in combination. This approach is different from ours because it is strictly focused on the increase of reusability, while we intend to support a wider range of reengineering goals.

Detection of Refactorings. A recent paper [5] describes four metrics-based heuristics that can be used for the detection of the changes that took place during the successive versions of a software systems. What relates it to our work is on the one hand the fact that it is also a detection technique and on the other hand it is also a metrics-based approach. The heuristics proposed in that paper finds those parts of a system that are often refactored, and which are therefore supposed to have an unstable design. It is probable that such design fragments present some sorts of design-flaws. The disadvantage of this approach is that it requires several versions and it will find only those parts that are changing.

6 Conclusions and Future Work

In this paper we presented a general metrics-based approach for problem detection. Based on this approach, we have defined detection techniques for two well-known design problems: god-classes and data-classes. Each detection technique is described using a consistent format. The main features of our approach are:

- It is *systematic* and *repeatable*, because the succession of steps that must be taken is clearly defined and each detection technique is described using a consistent format.
- It is *scalable* because the detection techniques are defined so that they reduce and focalise the investigation area only on those design fragments that are suspected to be flawed.

- It is *highly language independent* because the metrics are computed on the design-model, which is language independent. The only component that is bound to the programming language is the one that extracts the design information from the source files.

We have examnined an industrial case-study using this detection method and the results reported in this paper show so far that this approach can be successfully applied. Although more experimental investigation is needed, we believe that the next case-studies will validate the approach.

Our future efforts will concentrate on following fronts. First of all we need to continue to validate the two detection techniques against other software systems. This is necessary, because these techniques are by their nature heuristical and therefore it is their proven practical applicability which makes them interesting. The second front is to define further metrics-based detection techniques for other design-flaws, mainly based on the "bad-smells" described in [7]. We are currently working on the definition and validation of new techniques tackling the detection of other common design-flaws.

References

[1] J.M. Bieman, B.K. Kang. *Cohesion and Reuse in Object-Oriented Systems.* Proceedings of the ACM Symposium on Software Reusability, April 1995.

[2] E. Casais. *Re-engineering object-oriented legacy systems.* Journal of Object-Oriented Programming, pages 45-52, January 1998.

[3] O. Ciupke. *Automatic Detection of Design Problems in Object-Oriented Reengineering.* Technology of Object-Oriented Languages and Systems - TOOLS 30, pages 18-32, IEEE Computer Society, August 1999.

[4] S.R. Chidamber, C.F. Kemerer. *A Metrics Suite for Object-Oriented Design.* IEEE Transactions on Software Engineering, 20(6), June 1994

[5] S. Demeyer, S. Ducasse, O. Nierstrasz *Finding Refactorings via Change Metrics.* Proceedings of the OOPSLA'2000, ACM Press, 2000.

[6] K. Erni. *Anwnendung multipler Metriken bei der Entwicklung objektorientierter Frameworks.* ISBN 3-931546-03-9, Krehl Verlag, Münster, 1996.

[7] M. Fowler *Refactoring: Improving the Design of Existing Code.* ISBN 0-201-48567-2, Addison-Wesley, 1999

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software.* ISBN 0-201-63361-2, Addison-Wesley, 1994

[9] I. Jacobson, F. Lindstroem *Re-engineering of Old Systems to an Object-Oriented Architecture.* Proceedings of the OOPSLA'91, pp. 340-350, ACM, 1991

[10] R. Martin. *Design Principles and Patterns.* http://www.objectmentor.com, 2000.

[11] J.A. McCall, P.G. Richards, G.F. Walters. *Factors in Software Quality, Volume I* NTIS AD/A-049 014, NTIS Springfield, VA, 1977

[12] A.J. Riel *Object-Oriented Design Heuristics.* ISBN 0-201-63385-X, Addison-Wesley, 1996.