Rule-Based System for Flaw Specification and Detection in Object-Oriented Programs

El Hachemi Alikacem[1] and Houari A. Sahraoui[2]

[1] Computer Research Institute of Montreal
405, Ogilvy Avenue, Suite 100, Montreal, QC, Canada H3N 1M3
el-hachemi.alikacem@crim.ca
[2] Département d'Informatique et de Recherche Opérationnelle. Université de Montréal
CP. 6128 Succ. Centre-ville, Montréal, QC, Canada H3C 3J7
sahraouh@iro.umontreal.ca

Abstract. Low quality design leads to error-prone software that is difficult to understand, maintain and evolve. Thus, in order to improve its quality, software should be continuously inspected by examining the source code to identify potential flaws. However, this task can become complex and time-consuming when dealing with large size programs. To provide support for source code analysis, we propose a rule-based approach that allows the specification and detection of flaws. This approach provides a new language to describe flaws as sets of rules. The latter are translated into Jess's rule format, and given as input to Jess inference engine. The current work is an extension of our source code analysis platform and PatOIS, a metric description language. A main advantage of this approach is its extensibility since the tool is not limited to a set of predefined flaws. Existing flaws could be adapted to a specific context and new ones could be added.

Keywords: program inspection, flaw detection, anti-pattern, metrics, rule-based system.

1 Introduction

Flaws in software have a negative impact on many quality attributes such as maintainability, understandability, and reliability. Moreover, they hinder software evolution, since they make it hard for developers to perform changes. Source code inspection allows the identification of such flaws. However, this task is tedious and time consuming when performed manually. Therefore, tools are needed to support users for detecting flaws.

Numerous quality rules have been proposed in the literature [1,2,3,4,5] resulting from years of experience in software development. These rules define "best practices" for designing and implementing flaw-free object-oriented programs. Thus, flaw detection consists of the identification in the source code of situations corresponding to violations of these rules.

Several approaches have been proposed to detect flaws (details are given in Section 5). However, most of them do not offer flexible mechanisms to detect new flaws, or to adapt the detection of existing ones to the context of the analyzed program. This limitation may have a negative impact on the quality of the detection i.e., large number of false positives and a bad recall. In order to circumvent the above limitation, we propose an approach for flaw detection based on two main mechanisms: (1) a flaw specification mechanism which is used to define flaws as set of rules, and (2) a detection mechanism which includes an inference engine.

This approach is implemented on top of our platform of object-oriented source code analysis [6]. The architecture of the whole system is depicted in Figure 1. It is composed of three components: (1) the source code representation component, which is responsible for building a model representing the source code, (2) the metric extraction component, which is based on a metric description language called PatOIS, and (3) the flaw detection component. The latter takes as input a flaw specification and produces a rule-based system. To specify flaws, we developed a simple language, which is an extension of PatOIS, our metric description language.

This paper is organized as follows. In Sections 2 and 3, we summarize respectively the two existing components of the platform: the object-oriented program representation and the metric extraction. Our detection approach is detailed in Section 4. Related work is discussed in Section 5. Concluding remarks are given in Section 6.

2 Object-Oriented Program Representation

We defined a meta-model that includes OO programming key concepts. Indeed, this meta-model is designed to support typed object-oriented programs written in different programming languages (e.g., C++, C#, Eiffel, and Java). In the design of the meta-model, we consider three parts, each corresponding to a category of concepts:

1. **Common concepts** for which the syntax and the semantics are similar for all supported languages. These are the common concepts of object-oriented languages such as classes, methods and attributes.
2. **Variable concepts** with similar syntax but variation in semantics. A well-known example of this category is inheritance. Indeed, different languages have different operational semantics of inheritance with, in particular, different method-lookup strategies. To circumvent this problem, the semantic of these concepts is interpreted and explicitly represented during the mapping of the source code.
3. **Specific concepts** that exist only in a particular language. Language specific concepts are integrated in the meta-model explicitly. For example, Entity represents the notion of abstraction in the different OO languages (Fig. 2). It is specialized into Interface, Union, or Template that are present in Java or C++. In other words, we simply include in the meta-model the specific concepts of each language we consider.

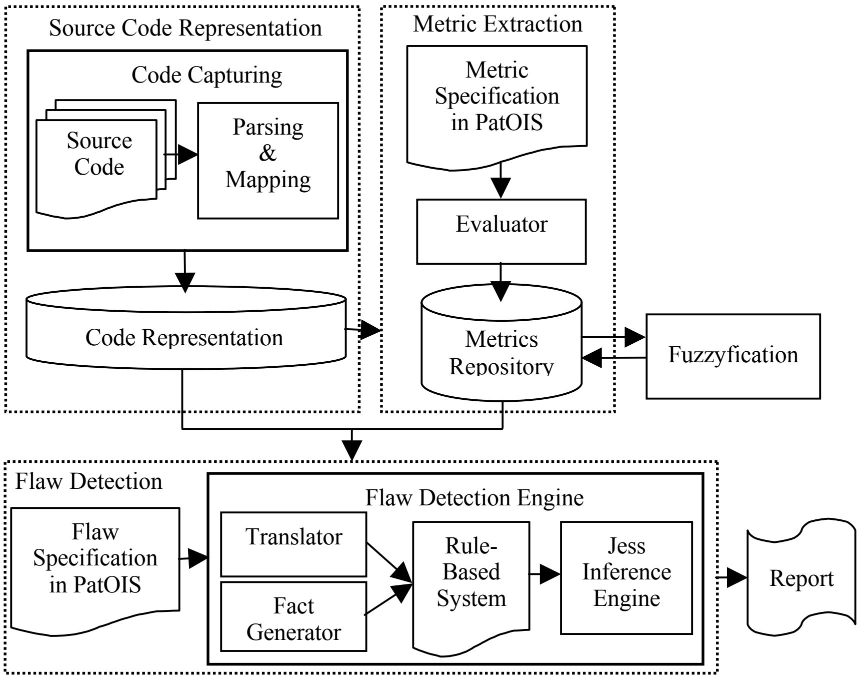

Fig. 1. Architecture of the flaw detection system

Considering the three categories of concepts presented above, we obtain a generic meta-model in which both syntactical constructs and semantic aspects are mapped.

As it is presented in Figure 1, the code representation conforming to the meta-model is generated by the module *parsing and mapping*. Thus, in order to adapt the framework to a particular programming language, a module should be implemented specifically for this language. This module is responsible for identifying the three categories of concepts and mappings them according to their nature.

3 Metric Description and Extraction

The metric extraction component collects product metrics that are calculated from the source code. These include design metrics (e.g., design coupling metrics). Extraction uses information derived from the generic representation. Therefore, to express a metric, we need mechanisms able to access data in the representation and perform operations on them. To do so, we designed the metric description language, named PatOIS (for Primitives, Operations, Iterator, and acceSsors), whose features are summarized in the remainder of this section. A detailed description could be found in [6]. Defining a dedicated language instead of using existing technologies such as OCL, with which PatOIS has similar features, or SQL, is motivated by providing users with a simple and expressive mechanism to define metrics.

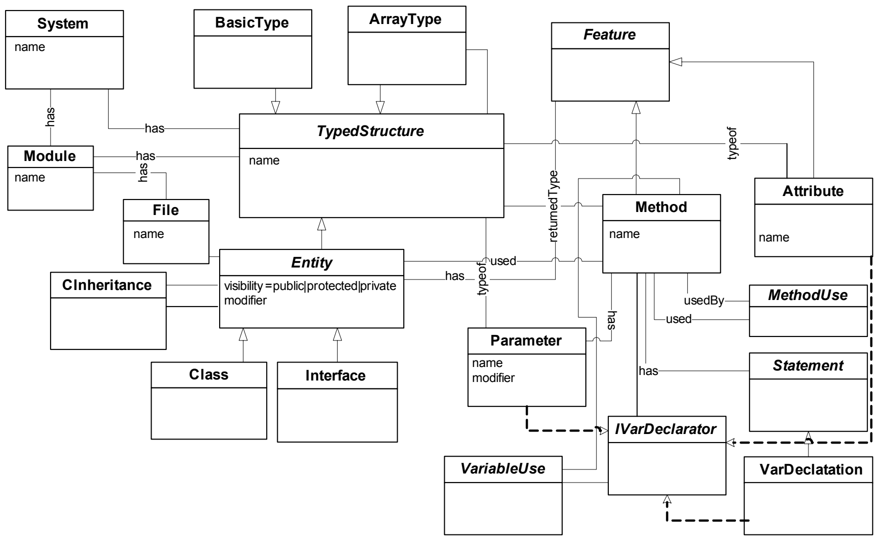

Fig. 2. Source code representation meta-model

The basic features of PatOIS are Primitives. These are procedures that allow selecting the base sets from the code representation. Primitives are hard-coded and are used as library functions when describing metrics. Three types of primitives are defined. The basic ones return a set of instances for a given concept in the meta-model. *classes*(), *interfaces*(), and *methods*() are examples of primitives that select the instances of respectively concepts Class, Interface, and Method for a particular program. The second type of primitives returns a set of the elements related to a specified element by a particular relationship. Examples of these primitives are *methods*(c) and *children*(c) that give respectively the set of methods of a class c, and the subclasses of c. The third type of primitives is the user-defined ones. They are implemented either for complex calculation or to reuse recurring descriptions. Examples of these metrics are the calculation of the distance between a class c and root/leaf classes in the inheritance tree.

The second important feature is Operations. Three categories of operations can be used: arithmetic (sum, minus, max, min, etc.), comparison (between numbers such as == and >, or between strings), and set (union, intersection, etc.) operations. A particular and useful operation is cardinality that calculates the size of a set given as a parameter. For instance, |*classes*()|.

Property access is the third category of features. They allow to access element properties. For example, the operation c.visibility returns the visibility of the class c.

The last feature is Iterator. It enables the manipulation of elements of a set. The simplified syntax of this operator is:

forAll (x : inputSet ; condition_clause ; SET AssignOperator expression)

For each element x of *inputSet*, if the condition (*condition_clause*) is true, then the predefined variable SET receives the value of *expression*. The returned value is the set of elements in SET at the end of the iteration. For example, the set of public classes can be obtained by iterating on the elements of the primitive classes() with the condition that the property visibility is equal to public. Formally:

$$forAll\,(c : classes\,();c.visibility\ == \ public\,;SET += c)$$

The language has a very few simple syntactic constructs. It does not require an important learning effort. It has been used to implement the commonly-used metrics such as complexity, inheritance, cohesion and coupling metrics with very few primitives and compact descriptions.

4 Flaw Detection Method

As stated in Section 1, numerous quality rules have been proposed in the literature. However, these rules are from different natures and concern different software aspects. They also have different levels of abstraction. Indeed, some of them are very precise (e.g. it is better to have less than 6 parameters for a method) while others are more general and correspond usually to guidelines and principles (e.g. a class should represent one abstraction). Due to this variety in the nature of the rules, different terms are used in the literature. The term design heuristic, which corresponds to principles and guidelines for implementing good quality programs, is used by Riel [5] and Marinescu [3]. Brown et al. [2] published the first book gathering flaws, in which these flaws are designated as anti-patterns. The latter are defined as structures that seem to be good solutions but, which backfire badly when applied. The concept of code smells, introduced by Fowler and Beck [7], are low-level design defects in the source code of a program. In the remainder of this paper, we use the generic term of *quality rules*.

4.1 Quality Rule Quantification

Several of the proposed rules are expressed in terms of metrics that are collected from the source code; thus, the detection of rule violations could be easily automated. However, there are often informal rules defined at a high level of abstraction. For such rules, a modeling effort is required in order to express them in quantifiable expressions. To model these rules, we use the GQM paradigm.

The GQM paradigm, proposed by Basili et al. [8], provides a support for the selection of metrics that should be collected in a software measurement activity. Three steps are involved in the GQM paradigm:

1. List the main goal and eventually its sub-goals.
2. From each goal or sub-goal, derive the questions that must be answered to determine if the goal/sub-goal is met.

3. Identify which metrics are necessary to answer to the different questions.

The GQM paradigm is an abstract method that has to be instantiated for each rule. To illustrate how this paradigm is used, let's consider the Blob anti-pattern. A Blob is a large controller class that depends on data stored in related data classes. Such a class implements a large part of the system's behaviour. The symptoms of the Blob include a large class with a low cohesion, and small associated data classes.

The first step is to identify the goal and sub-goals of the detection. The anti-pattern symbolizes the goal. Then, based on its description, sub-goals are isolated following a refinement process. In the final step, metrics are identified for each sub-goal. For the Blob anti-pattern, we obtain the model in Figure 3.

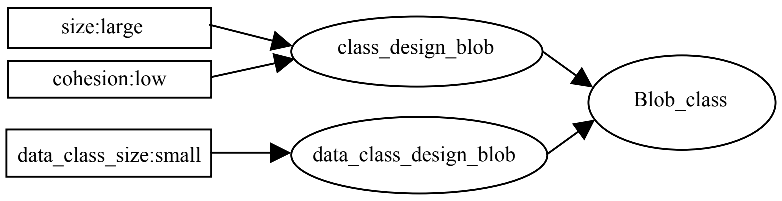

Figure 3. Blob anti-pattern model

The Blob's model could be used to detect Bob occurrences as shown in the following set of informal rules:

> ***if*** *a class is large* ***and*** *its cohesion is low* ***then*** *class_design_blob*
> ***if*** *data_classes are small* ***then*** *data_class_design_blob*
> ***if*** *class_design_blob* ***and*** *data_class_design_blob* ***then*** *Blob_class*

Our approach offers a detection mechanism which supports both threshold-based and fuzzy-label-based premises. For the first case, in the flaw specification, metrics corresponding to quantifiable properties are used directly. For instance, the metric number of attributes (NOA) can be used to measure the class size, and the lack of cohesion metric (LCOM) to measure the class cohesion. In addition, a threshold is defined for each metric. For example, a threshold is needed to indicate when a class is large. One can assume that a class that contains more than 20 attributes is a large class. This first case corresponds to the state of the art.

For the second case, our approach supports fuzzy values as thresholds. A fuzzy value is a label obtained from the precise value of a metric by applying a fuzzification process which is based on predefined membership functions.

To illustrate this process, let's consider the NOA metric and the membership functions shown in Figure 4. If the number of attributes of a class NOA is 19, then the corresponding fuzzy value is large with a degree of truth 0.8. Moreover, the class is also medium with a degree of truth 0.2.

The fuzzy values relax the strict thresholds on the metrics, and thus improve the precision of the flaw detection. Indeed, a class that contains 19 attributes (NOA = 19) is not a large class when compared to the threshold 20. On the other hand, the same class is considered as a large class with a fuzzy value corresponding to its NOA metric.

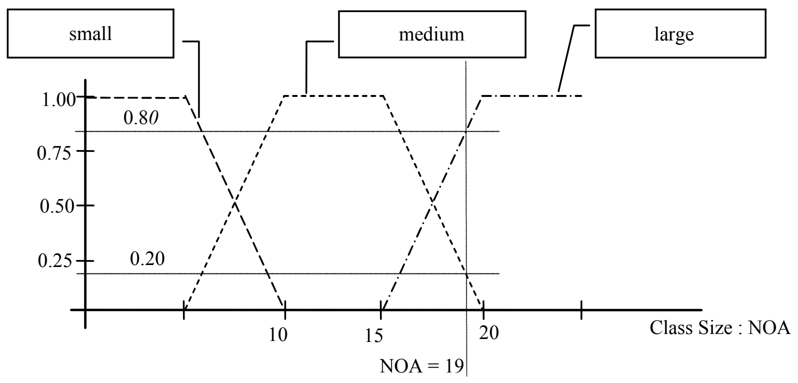

Fig. 4. Membership functions corresponding to NOA

4.2 Flaw Specification Language

In the previous section, we provided an informal specification of the Blob anti-pattern. For an automatic detection, flaws should be specified in a formal way. Therefore, we defined a flaw specification language that allows the specification of flaws as a set of rules. The following grammar describes the main part of the specification language:

```
rule_def      ::= RULEDEF ruleName { rule_desc } (rule_ctxt)?
rule_desc     ::= IF rule_cdt THEN rule_concl
rule_cdt      ::= (expression)
rule_concl    ::= IDENTIFIER | fuzzy_Expr
rule_ctxt     ::= CONTEXT { (ctxtOpt)+ }
ctxtOpt       ::= APPLYTO    : entity
                | TARGET     : connectivity
                | QUANTIFIER : label
entity        ::= SYSTEM | MODULE | CLASS | METHOD
connectivity  ::= ASSOCIATION|INHERITANCE
label         ::= ALL | ONE | MOST
expression    ::= IDENTIFIER
                |  binary_expr
                |  unary_Expr
                |  fuzzy_expr
fuzzy_expr    ::= IDENTIFIER ~ IDENTIFIER
binary_expr   ::= expression operator expression
unary_expr    ::= ! expression
operator      ::= +|-|*|/|<|>|<=|>=|==|!=|&&
```

A rule is identified by the keyword RULEDEF, followed by a name, a rule description, which is an if-then construct, and a rule context that defines the entities e.g., classes, or methods, on which the rule is applied. We illustrate in the following paragraph how this specification language is used with two examples.

The first example is the identification of large classes with low cohesion. We use NOA and LCOM metrics to measure respectively the class size and the cohesion. The following code specifies the bad implementation of a class:

```
ruleDef badImplClass {
   if (NOA > 20 && LCOM >3) then bad_impl_class ~ high
} context {
   applyTo : class;
}
```

Fuzzy values might be used instead of precise thresholds as it is shown if the following code:

```
ruleDef badImplClass {
   if (NOA ~ large && LCOM ~ high) then bad_impl_class ~ high
} context {
   applyTo : class;
}
```

In both specifications, for each class when the condition is evaluated as true, the conclusion is generated as a new fact for this class. In addition, a degree of truth is calculated for this new fact. Actually, it equals to the degree of truth of the condition clause which is the degree of truth of its term or the product of the degrees of truth of each term of the condition clause. Note that the degree of truth of a term which is not based on a fuzzy value e.g., NOA > 20, is 1.

Usually, membership functions are defined for each variable e.g., NOA and LCOM metrics. Even when same labels are used, they correspond to different values since they are derived from different membership functions.

However, membership functions are not defined for non-terminal members, for instance `bad_impl_class` defined in `badImplClass` rule. The label defined for non-terminal members is used in the generation of facts when the rule is triggered.

The second example is the identification of all the classes associated with small classes.

```
ruleDef dataClassDesign {
   if (NOA ~ small) then data_class_blob ~ high
} context {
   applyTo : class;
   target  : association;
   quantifier : all;
}
```

The main difference between the `badImplClass` rule and `dataClassDesign` rule is the context. In the second rule, the connectivity property indicates that the condition is evaluated on the associated classes of each class. Moreover, the conclusion is evaluated as true if the condition is true for all (because of the `quantifier` property defined in the context) the associated classes. In other words, a new fact `data_class_blob` is generated for each class for which the condition is evaluated as true for all its associated classes.

The additional following rule, based on the two previous ones, specifies the Blob anti-pattern:

```
ruleDef blobClass {
 If (bad_impl_class ~ high && data_class_blob ~ high) then
     blob_class ~ high
}
```

In this third rule, the condition is evaluated as true when the two facts for the same class are already generated by the two first rules. As a result, the fact `blob_class` is generated and corresponds to the expected result.

4.3 Flaw Detection

The flaw detection engine, as presented in Figure 1, contains two modules, *Translator* and *Fact Generator*, and uses Jess inference engine [9]. *Translator* module translates the flaw specification rules in PatOIS to Jess's rule format. Fact generator generates the facts corresponding to the entities on which the rule is applied to. For illustration purpose, let's consider the following simple rule in PatOIS:

```
ruleDef badImplClass {
   if (NOA > 20) then bad_impl_class ~ high
} context {
   applyTo : class;
}
```

The corresponding generated Jess rule is:

```
1:   (deftemplate datafact (slot entity) (slot property)
                           (slot value)(slot dtruth))
2:   (deftemplate nonTerminal (slot label) (slot value)
                              (slot entity)(slot dtruth))
3.1: (defrule badImplClass (datafact
        (property ?prop&:(= ?prop "NOA"))
        (value ?val&:(> ?val 20))
        (entity ?eName)(dtruth ?dt))
     =>
3.2: (assert (nonTerminal (label bad_impl_class)
             (value high) (entity ?eName)(dtruth ?dt)))
```

Lines 1 and 2 correspond to the templates definition of facts generated by the *Fact Generator* module, and facts generated during the rule inference, respectively. Line 3 is a rule definition which is composed of a condition clause (Line 3.1) and a conclusion clause(3.2).

For each fact that matches the condition clause, Jess inference engine executes the conclusion, which generates (assert operator) a new fact according to the template defined in line 2.

However, before launching the inference engine, Jess's working memory should be populated, which is done by the *Fact Generator* module. The latter uses data from the code source representation and the metrics repository to generate the facts.

For instance, if the code source representation contains two classes: `MyClass1` and `MyClass2`, and their corresponding NOA metrics in the repository are 7 and 29, then the two following facts are generated:

```
(datafact (entity MyClass1)(property NOA)(value 7)(dtruth 1))
(datafact (entity MyClass2)(property NOA)(value 29)(dtruth 1))
```

Considering the rule `badImplClass`, only the second fact matches with rule's condition; therefore, a new fact is generated into the working memory:

```
(nonTerminal (label bad_impl_class)(value high)
             (entity MyClass2)(dtruth 1))
```

The detection results are represented by the facts in which the value of the member `label` is the non-terminal term in the conclusion clause of the rule describing the flaw, `bad_impl_class` or `blob_class`, for instance.

5 Related Work

During the past decade, different tools have been developed using various techniques and approaches. Marinescu [3] proposes metric-based strategies to detect flaws at method, class, and system levels, implemented in the iPlasma tool. The detection strategies consist of combining metrics with a set of operators and using both absolute and relative thresholds. Compared to our fuzzy thresholds, relative thresholds are still unique values with binary decisions.

Moha et al. [10] propose the method DECOR for the specification and detection of code smells. They specify flaws using sets of rules, then generate detection algorithms automatically as Java code. In comparison to our approach, their specification language is complex and necessitates skills in programming; therefore, the extension to new defects may need an important effort. Another difference is the metric extraction mechanism, which is embedded in both platforms. With our platform, metrics can be easily added or adapted in order to define flaw specifications precisely, which is not feasible in DÉCOR platform.

Some approaches use query mechanisms. CROCOPAT [11] provides a simple and expressive query language to manipulate relations of any arity, including graphs. However, this language is at a low-level of abstraction and requires software engineers to understand the implementation details of the underlying model of CROCOPAT.

Tools such as PMD [12], CHECKSTYLE [13], and FXCOP [14] detect problems related to coding standards, bugs patterns or dead code. These tools focus on code-level problems but do not address higher-level design defects such as anti-patterns. The extension to new defects, when it is allowed, is achieved with a programming language such as Java.

Bär et al. [1] describe heuristics with queries in Prolog. In this formalism, only structural properties of a source code are specified without considering metrics.

In comparison with previous work, our approach is characterized by a simple and high-level specification language that enables users to manipulate structural information in conjunction with metrics to specify and detect flaws.

6 Conclusion

Flaw detection is important to improve the quality of programs and facilitate their evolution. Therefore, tools are essential to support developers in code inspection tasks. Such tools should have at least two characteristics: flexibility and detection precision. From the one hand, developers should be able to extend the tool to detect new flaws, and to adapt it to their context. From the other hand, they do not adopt tools that produce a significant number of false positives. Thanks to our flaw specification language, developers can easily extend and adapt the tool not only to the context e.g., selecting thresholds according to the program to analyze, but also refine the specification e.g., adding additional terms in condition clauses or even new rules, in order to improve the quality of the detection. Moreover, handling fuzzy values is a significant feature in the improvement of the detection quality. In the current state of the implementation progress, membership functions are specified manually for each metric involved in the specification. Later, this process will be automated using clustering approach. Such process is presented in [15].

In order to evaluate the expressiveness of our specification language, we performed preliminary experimentations on small size programs. The obtained results are very encouraging. We are planning to conduct a large-scale validation.

References

1. Bär, H., Ciupke O.: Exploiting Design Heuristics for Automatic Problem Detection. In *S.* Ducasse and J. Weisbrod editors, Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering, number 6/7/98 in FZI Report, (June 1998).
2. Brown, W. J., Malveau, R. C., McCormick, H. W., and Nowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Project in Crisis. John Wiley & Sons, (1998).
3. Marinescu, R. Detection Strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th ICSM Conference, (2004), 350-359.
4. Martin, R. C. Design Principles and Patterns. Object Mentor, www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf. (2000)
5. Riel, J. A.: Object-Oriented Design Heuristics. Addison Wesley, (1996).
6. El Hachemi Alikacem, Houari A. Sahraoui: A Metric Extraction Framework Based on a High-Level Description Language. In: SCAM, pp.159-167, 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009.
7. Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, (1999).
8. Basili V.R., Rombach H.D.: The TAME project: Towards improvement-oriented software environments. In,: IEEE Transactions on Software Engineering, 14(6), 1988.
9. Jess: A rule inference engine: http://www.jessrules.com/
10. Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur: DECOR: A Method for the Specification and Detection of Code and Design Smells. In: IEEE Transactions on Software Engineering. January/February 2010 (vol. 36 no. 1), pp. 20-36, ISSN: 0098-5589
11. Beyer, D., Noack, A., and Lewerentz, C.: Efficient relational calculation for software analysis. In: Transactions on Software Engineering, 31(2), (Feb. 2005), 137–149
12. PMD: http://www.sourceforge.net
13. CheckStyle, http://checkstyle.sourceforge.net (2004).
14. FXCop: http://www.gotdotnet.com/team/fxcop (2006).
15. Malak, G., Sahraoui, H.A., Badri, L., Badri, M.: Modeling web quality using a probabilistic approach: An empirical validation. To appear in ACM Trans. Web (2010).