UML-based Design Test Generation

Waldemar Neto
Department of Computer Science, Federal University of Campina Grande
58.109-970, Campina Grande, PB, Brazil
neto@dsc.ufcg.edu.br

João Arthur
Department of Computer Science, Federal University of Campina Grande
58.109-970, Campina Grande, PB, Brazil
jarthur@dsc.ufcg.edu.br

Franklin Ramalho
Department of Computer Science, Federal University of Campina Grande
58.109-970, Campina Grande, PB, Brazil
franklin@dsc.ufcg.edu.br

ABSTRACT

In this paper we investigate and propose a fully automated technique to perform conformance checking of Java implementations against UML class diagrams. In our approach, we reused the DesignWizard Java API that allows us to write design rules as JUnit tests, i.e., to write them as code directly in the programming language. We fully pursued MDA as the approach for generating the design tests and hence we used several MDA artifacts, such as metamodels, models and transformations. A proof of concept of the technique has been implemented and evaluated. We performed several experiments on simple scenarios. Simple designs involving classes, associations, inheritance have been checked. Compared to previous related work, the advantage of our approach lies in the fact that we automatically generate design tests from UML class diagrams to Java code that play the dual role of design test and implementation language. Thus, we check the conformance between the design and the implementation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms

Design, Reliability, Verification

Keywords

Design Test, code generation, MDA, UML Class Diagram

1. INTRODUCTION

Design conformance checking, or conformance checking for short, is the process of checking whether an implementation complies with a given design. It is especially relevant in development processes that promote a clear distinction of design and implementation activities like RUP. And, in particular, if different teams develop design and implementation. Even in agile processes, some level of conformance checking is necessary to keep implementation in shape with design and architectural rules or guidelines. In current practice, however, conformance checking is performed manually through code and design reviews (or, at some extent, with pair programming in the case of agile processes). In this paper we investigate and propose a fully automated technique to perform conformance checking of Java implementations against UML class diagrams.

In a previous work, we developed the concept of design tests [7]. A design test is an automated test that checks conformance of implementation code against an explicitly programmed design rule (in Section 3 we detail the concept). We developed a simple to use API, named DesignWizard [7], which allows us to write design rules as JUnit tests. In such approach, design rules must be written as code directly in the programming language. It can be very attractive and adequate to programmers and agile teams because it allows the application of the technique in an incremental and partial way. It is incremental because the designer needs not write a big up front design - he can evolve from a few architectural rules or no rule at all to a completely detailed low level design. It is partial because it does not impose that all aspects of a design be specified/checked - one can decide to focus on critical aspects of the design and ignore all others completely. There are development processes, on the other hand, in which design and implementation are not so close. In this case, design models are considered as mandatory specifications that must be followed during implementation. In this scenario, however, the code based technique of design tests to write design rules can be cumbersome because it implies the duplication of work: all rules must be derived from the models. Clearly, it also introduces the problem to keep consistency between design models and code based design rules.

In this work we propose an MDA-based technique to automatically generate design tests from design models. We specifically concentrate on Java implementations and UML models. The technique allows the designer to develop plain UML models and derive design tests to check implementations against the given design. Whenever the design evolves, all design tests can be fully regenerated with no cost, eliminating the effort required to synchronize artifacts.

Another important property of this approach is that no additional language is required in the process to express design rules. Even if customized rules are needed to further adjust the conformance checking, they can be entirely writ-

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

ten in Java.

A proof of concept of the technique has been implemented and evaluated. We performed several experiments on simple scenarios. Simple designs involving classes, associations, inheritance and specified attributes and methods have been checked. We have been able to generate design tests to check the design of small applications. Both conforming and not conforming implementations have been fully automatically verified against their UML specifications using DesignWizard with no effort to write code based design rules. Despite the simplicity of the scenarios, the architecture of the tool is flexible enough to be evolved to cope with further aspects of UML. In fact, it can be evolved to include customized understandings of UML specifications to support different expectations of development teams like OCL constraints.

It is important to observe that our approach does not validate, verify or test the UML models. It considers that the models are correct and that the implementation must reflect the models. In other words, it checks whether the code is in conformance with the previously created design.

In Section 2 of this document we elaborate on the concept of design tests and the DesignWizard API that supports it, showing a few examples of design rules. In Section 3 we go into details of how code for DesignWizard is derived from UML models using an MDA-based technique - we cover the concept of MDA, the architecture of the tool, details of the transformation rules. In Section 4 we relate our work to existing approaches to code conformance and design checking. We conclude our paper in Section 6.

2. DESIGN TEST

Unit test examines the behavior of a distinct unit of work. Within a Java application, the distinct unit of work is often (but not always) a single method. This kind of test often focuses on testing whether a method is following the terms of its API contract [17]. The fact of the code has been tested using unit test does not mean that its design is in conformance with the specified design. The problem of testing the design has not been well addressed. There is no natural way to verify the design an automated manner.

When people make changes in the code without a full comprehension of its design, the code loses its structure. It becomes harder to see the design by reading the code. Loss of code structure has a cumulative effect. The harder it is to see the design in the code, the harder is to preserve it, and the more rapidly it decays [11]. For this reason, it is necessary to create a mechanism to accomplish automatically the verification of the design code.

To address the problem described above, we present a test based approach that assures that a program conforms to its design/architecture. The approach is based on the assumption that writing design rules in the target programming language is appealing to developers as it has an attractive learning curve and can be easily understood by both designers and programmers. We also show that the approach can be integrated with most development processes, because conformance checking is performed during test execution and no additional activities must be added to the process.

The key concern of our approach is the construction of the design test, which is a kind of test for specifying the desired design of the code to be implemented. Design rules must be described considering structural information about the entities of the code and their relationships. The main goal of constructing design tests is to check code conformance using software testing, assuring quality in software evolution and avoiding software erosion.

A simple example of a design test is one that specifies the desired behavior for communication between classes. Let A and B be classes from a given project. Assume that the designer from the project does not want to allow that class A uses class B, that is, there must not exist any call of method or access to any attribute from class B in class A. If this restriction is implemented as a test, the designer can find automatically, by executing the design test, any violation of the specified rule.

We advocate that by adding design tests on the development process, such as XP [5] does with unit tests, improves the quality of software and the comprehension of the code by programmers. Keeping the structure of the code in conformance with the specified design assures that the code developed does not break any rule described as design tests. Hence, this process improves the reliability of the code.

In order to describe the expected structure of the design, a design test needs to know structural information about relationships of the entities from the code to be tested.

2.1 Design Wizard

Design Wizard (DW) is a library that provides a powerful Java API which offers services to acquire structural information about the code. For instance, it provides which methods from code are static. DW gives support to the designer on the task of composing design tests without adding any effort to learn a new language to specify the expected design. Design tests can use the API from DW to collect information about entities from the code and use them to test its structure.

In order to achieve static analysis of the code, DW reads the classes of a java code by using the ASM bytecode manipulation framework [6]. The information about the structure of the code is extracted and modeled in classes, methods, fields and their relationships.

The usage of DW is very simple, it is only necessary put the designwizard.jar on the classpath, compose the design tests and execute them. For this reason, the tool is entirely independent of any Integrated Development Environment (IDE) or platform. We have already tested its behavior on Eclipse and NetBeans. And both on UNIX and Windows operating systems.

The process of testing the design happens as follows. The designers compose the design test with support from DW API. In the meantime, programmers are coding the systems. Then, the tests are executed to verify whether the code wrote by the programmers is in conformance with the design test. In this case, the framework JUnit [16] has been used to run the tests and report failures. The output of the verification is the same output from JUnit framework, that is, failures are reported with red bar while the successful is reported with a green bar.

Let us get back to the example of previous section and try to write it with support of DW and JUnit framework. There is a simple and natural way to compose a test for the restriction of the designer by using the API from DW, as shown in Code 1. For instance, in this Code we can see some methods:(1) DesignWizard("project.jar") at line 3;(2) getClass("A") at line 5;(3) getClassesUsedBy() at line 7.

Line 3 creates a new instance of DW passing as a param-

eter the path of the jar file from the project to be tested. At this moment, DW extracts structural information from the classes within the jar file. At line 5, the method getClass("A") is called and returns the representation for class A. With the object that represents class A (clazz), the designer now can invoke the method getClassesUsedBy() to list all classes that are used by A on the code tested. After that, the designer must only to make sure that class B is not within the set of classes called by class A.

Code 1: DesignTest

```
1 public void testCommunication() {
2   DesignWizard dw;
3   dw = new DesignWizard(''project.jar'');
4   designwizard.ui.Class clazz;
5   clazz = dw.getClass(''A'');
6   Set<String> usedBy;
7   usedBy = clazz.getClassesUsedBy();
8   assertFalse(usedBy.contains(''B''))
9 }
```

3. STRUCTURE

3.1 MDA

MDA is a software engineering approach defined by the OMG. Its represents just one view of MDD (Model-Driven Development), though it is perhaps the most prevalent at present [13]. The key idea of the MDA is to shift the emphasis in effort and time during the software life cycle away from implementation towards modeling, meta-modeling and model transformations. In order to reach this goal, MDA prescribes the elaboration of a set of standard models. The first one is the CIM (Computational Independent Model) that captures an organizations ontology and activities independently of the requirements of a computational system. The CIM is used as input for the elaboration of the PIM (Platform Independent Model), which captures the requirements and design of a computational system independently of any targets implementation platform. In turn, the PIM serves as input for the elaboration of the PSM (Platform Specific Model), a translation of the PIM geared towards a specific platform. Finally, the PSM serves as input to the implementation code.

The MDA potential for automation is fully reached through transformations that must be specified between the above mentioned models. In this sense, there are a set of transformation languages, such as QVT [20] or ATL [2], supported by their respective transformation engines responsible for executing them.

In this work we pursued MDA as the approach for generating design test because: (1) UML models are in fact PIMs; and (2) Java or JUnit code are in fact PSMs. Thus, by reusing the MDA standards and frameworks (widely used at the industry and academy) and additionally by specifying the transformations from (1) to (2), we take profit from the MDA benefits aforementioned.

3.2 Architecture

We propose a MDA-based architecture for automatic generation of design test with DW. An overview of this architecture is shown in Figure 1.

The architecture consists of pivotal MDA elements that are briefly described as follows:

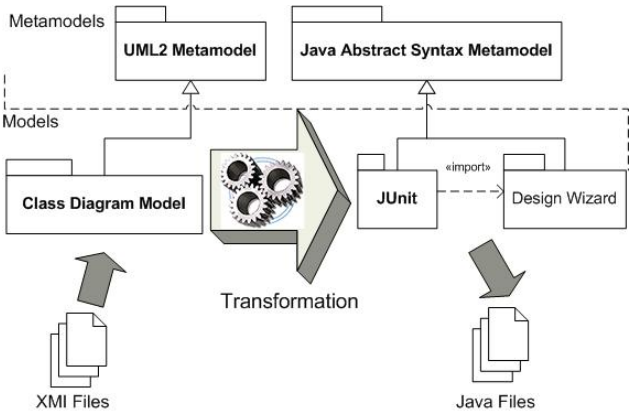

Figure 1: *The propose MDA-based Architecture.*

- **Metamodels**: The MDA transformations are based on metamodels that describe how a model must be formed. Since we generate Java/JUnit code from UML models, two metamodels are required: (1) source UML metamodel stated at top left in Fig. 1; and (2) target Java metamodel, at top right in Fig. 1. We have reused both. The former is provided by the OMG consortium [21]. The latter is the Java Abstract Syntax (JAS) metamodel [14], a complete Java code representation fully adopted by the Eclipse IDE;
- **Models**: The models are instances of metamodels. In our architecture, we have two kinds of models: UML class diagrams and Java/JUnit models. The former is the input model and the latter is the output model;
- **Transformations**: Rules that specify how the input models are transformed to target models according to their respective metamodels.

In order to build and validate our approach according to the architecture shown in Fig. 1 we have adopted the ATL-DT [4] framework. It is an Eclipse plugin that allows in an integrated way: (1) specifying and validating metamodels; (2) creating and validating models in conformance with their provided metamodels; (3) specifying and executing model transformations based on metamodels and applied to valid models. ATL-DT supports a set of formats, including the standard proposed by the OMG, such as: XMI (XML Metadata Interchange) for exchanging metadata information; and MOF (Meta Object Facility), for specifying metamodels. By adopting a framework that fully supports most of the OMG standards, we facilitate the integration of our work with existing tools, such as model or code editors.

It is important to emphasize that following a MDA approach, transformations may be defined from any kind of UML diagram specified at the design phase. This paper is focused on the transformations from UML class diagrams to Java/JUnit code, as shown in Fig. 1.

3.3 Transformation Rules

Although QVT [20] is the current OMG's purpose for specifying transformations in the MDA vision, we have adopted ATL for specifying our transformations. This is due main because QVT tools have still low robustness and its use is not widely disseminated. On the other hand, ATL has a

framework widely used by an increasing and enthusiast community, with full support to model operations, where, in an integrated way, one can specify and instantiate metamodels as well as specify and execute transformations on them. In addition, ATL has a wide set of transformation examples available at the literature and discussion lists, in contrast of QVT, whose documentation is poor and not didactic.

Therefore, our transformations from UML class diagrams patterns to the Java/JUnit patterns are implemented as ATL rules. In essence, they are rewrite rules that match input models patterns, also expressed as UML class diagram elements on the input metamodel and produce output patterns, expressed as Java/JUnit patterns.

Our set of ATL transformation rules generates design tests to verify some UML class diagram artifacts, such as classes, generalizations and associations. However, due to lack of space, we show in this paper details about the design test generation only for simple classes.

Consider the UML class pattern shown in Fig. 2. From each occurrence of this class pattern, we must generate a design test pattern as shown in Code 2.

In fact, the first six lines shown in Code 2 are responsible for initializing and configuring the design wizard and are not dependable of any element from the source metamodel. This test verifies the attribute and method signatures of the matched class.

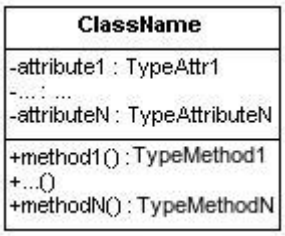

Figure 2: *UML Class Pattern.*

We show in Code 3 an excerpt of the ATL transformation named ClassDesignTestGeneration that is able to automatically generate the design test code shown in Code 2 from the UML class diagram shown in Fig. 2. In this rule, lines 1-2 define the name of the ATL transformation module (one ATL module may contain one or more ATL rules) and the input and output model variables, respectively. Line 3 defines the name of the ATL rule, whereas line 4 states that the Class element is that one from the source UML metamodel to be transformed. The remaining lines specify which Java/JUnit elements have to be transformed to. Due to lack of space, we just show how to generate the target elements shown at lines 5 of Code 2 that indicate the execution of a specific method of the DW framework. This method verifies the existence of a given class whose name is taken as parameter. It returns an associated ClassNode instance if the parameter is an existing class. Otherwise, it throws an exception InexistentEntityException.

Lines 7-37 create the whole assignment from the (1) method invocation getClass from the instance *dw* taking the input class name as parameter lines 25-37; to the (2) variable declaration statement declaring the variable c (lines 7-24).

Concerning the generalization test, we check whether the super class and their children classes exist. In addition, we verify if the children classes actually extend the super class, through the method isSuperClass of ClassNode.

Code 2: General Design Test for Class

```
1  public void testEntityClass() throws
2       IOException, InexistentEntityException{
3    DesignWizard dw = new DesignWizard(
4      ''/ClassRootFolder'');
5    ClassNode c = dw.getClass(''ClassName'');
6    String[][] attributes = {
7      { ''atrribute1'', ''TypeAttr1'' },
8      ...,
9      { ''atrributeN'', ''TypeAttrN'' } };
10   String[][] methods = {
11     { ''method1'', ''TypeMethod1'' },
12     ...,
13     { ''methodN'', ''TypeMethodN'' } };
14   for(int i=0; i>attributes.length; i++){
15     FieldNode n = dw.getField(
16       c.getName() + ''.'' + attributes[i][0]);
17     assertEquals(attributes[i][1],
18       n.getDeclaredType().getName());
19   }
20   for(int i=0; i>methods.length; i++){
21     MethodNode n = dw.getMethod(
22        c.getName() + ''.'' + methods[i][0]);
23     assertEquals(methods[i][1],
24        n.getReturnType().getName());
25   }
26 }
```

Code 3: Stretch Transforation to ClassNode

```
1  module UML2TestClass;
2  create J: JavaAbstractSyntax from U: uml2;
3  rule ClassDesignTestGeneration{
4   from class:uml2!Class
5  to
6  ...
7    declarationClassNode:JavaAbstractSyntax
8           !VariableDeclarationStatement(
9       type <- classNodeType,
10      fragments <- Set{classNodeVariable}),
11   classNodeType:JavaAbstractSyntax
12          !SimpleType(
13     name <- nameClassNode    ),
14   nameClassNode: JavaAbstractSyntax
15          !SimpleName(
16      identifier <-
17          'desingwizard.main.ClassNode'),
18   classNodeVariable:JavaAbstractSyntax
19          !VariableDeclarationFragment(
20     name <- simpleNameVarClassNode,
21     initializer <- classNodeCreator    ),
22   simpleNameVarClassNode:JavaAbstractSyntax
23          !SimpleName(
24     identifier <- 'c'     ),
25   classNodeCreator:JavaAbstractSyntax
26          !MethodInvocation(
27     expression <-nameVarDZ2NodeCreator,
28     name <- methodName,
29     arguments <- Set{parameterClassNode}),
30   nameVarDZ2NodeCreator:JavaAbstractSyntax
31          !SimpleName(
32     identifier <- 'dw'     ),
33   methodName:JavaAbstractSyntax!SimpleName(
34     identifier <- 'getClass'    ),
35     parameterClassNode:JavaAbstractSyntax
36          !StringLiteral(
37     escapedValue <- class.name      ),
38 ...
```

Concerning the association test, we verify the existence of the participating classes. However, since there is no direct Java counterpart for UML associations we follow the approach of Gonzalo et al. [12], because it is simpler and some important tools implement this approach [15]. According to this approach, an association in the input UML model (with or no aggregation and composition) is mapped to a Java attribute of the corresponding generated Java class. Thus, we verify whether there is an attribute with same name of the association role and additionally we verify whether this attribute has the same type of the class to which the association targets. For bi-directional association, this test must create this verification for both classes participating in the association.

Although the design tests are different, the MDA transformations for generalization test and association test are similar to the class test transformation illustrated above.

3.4 A Simple Verification Scenario

Our approach, detailed in previous sessions, was developed for general UML class digram. In order to illustrate its application, consider the class Person shown in Fig. 3. It has three 3 attributes: (1) String *name*; (2) String *id*; and (3) int *phone*. In addition, this class has two methods: (1) int *getName()*; and (2) boolean *fire()*.

By applying the rule ClassDesignTestGeneration, we fully automatically generate the class design test shown in Code 4. For instance, lines 9-10 in Code 4 are generated by applying the lines 7-37 of the DesingTestGeneration Transformation rule shown in Code 3.

The design tests generated from the Class Person arte shown in Code 4 where we illustrate design tests ranging class as well as its attribute and method signatures. Lines 5-6 configure the framework DW. Line 7 verifies the existence of the class Person - whether it does not exist the exception InexistentEntityException will be thrown. Lines 8-11 define the class attributes whose signatures are later checked at lines 15-20, whereas lines 12-14 define the class methods whose signatures are later checked at lines 21-26. In both cases, the properties to be checked are obtained from the UML class diagrams. For instance, in case of the attribute signature not being the same obtained from the UML model, an exception InexistentEntityException will be thrown at lines 16-17. In addition, in case of the class having an attribute with same name but different type from that specified on the UML model, the assert at lines 18-19 will fail.

Figure 3: *Class Person.*

4. RELATED WORK

Previous research related to our approach can be grouped in two lines of work. The first one focuses on generating design test from UML structural and behavioral diagrams. The second one advocates that design defects, named code smells [11], must be detected and removed from code.

Code 4: Design Test for Class Person

```
1  public class AllClassesInSuiteTest
2                          extends TestCase {
3    public void testClassPerson() throws
4        IOException, InexistentEntityException{
5      DesignWizard dw = new DesignWizard
6        (''/HumanResoucesProject'');
7      ClassNode c = dw.getClass(''Person'');
8      String[][] attributes = {
9        {''name'', ''String''},
10       {''id'', ''String''},
11       {''phone'', ''int''} };
12     String[][] methods = {
13       {''String'', ''getName''},
14       {''boolean'', ''fire''} };
15     for(int i=0; i>attributes.length; i++){
16       FieldNode n = dw.getField(c.getName()+
17         ''.''+attributes[i][0]);
18       assertEquals(attributes[i][1],
19         n.getDeclaredType().getName());
20     }
21     for(int i=0; i>methods.length; i++){
22       MethodNode n = dw.getMethod(c.getName()+
23         ''.''+methods[i][0]);
24       assertEquals(methods[i][1],
25         n.getReturnType().getName());
26     }
27   }
28 }
```

Following the first line of related work, Design Peer Review [1] has surged as one process widely used on software companies. It consists of a mechanism for ensuring design standards by separating the team in groups and making them analyze the code of each other. Design Review is a manual process that may lead the team to errors during the analysis. In addition, the process may take much time to complete, since analyzing several classes from a big project may take several hours. Another design test approach is defined by Trung T. et al.[8]. They generate inputs and exercises executable UML diagrams generated from class diagrams and activity diagrams. After that, it compares the expected behavior of a design with the detected behavior during the test execution. Then, failures are reported if the observed behavior differs from the expected behavior. This approach differs from ours due to the fact that they test the behavior of the program but not its structure. A similar approach is encoded in PTIDEJ [18], a tool suite that builds program representations from static and dynamic models of Java programs. The suite purpose is to accomplish identification of Design Patterns and Design Defects using a language and framework. However, the focus of this suite is on identifying design defects and encouraging the usage of them. Furthermore, it requires an additional intermediate language to express the design test.

Following the second line of related work, the Find Bugs [10] tool uses static analysis to automatically discover bugs on java code. It analyzes the bytecode of a java application and generates a report containing the probable bugs found on the analysis. Another similar tool LClint [9] has been used on code wrote in C language. As we can see, these tools focus on detecting low-level problems in source code such as possible null pointer references and unused code. However, we focus on design level and our purpose is to de-

tect whether the code is in conformance with a given UML design. The designer can implement a design test that specifies a good programming practice avoiding, through test, bad smells on code developed. In this line of work, Moha et al.[19] also proposes an approach to find design defects. To achieve this, they defined BNF grammar based language for specifying the defects to be found. They also developed a framework for automatic generation of design defects detection algorithms from the design rules.

It is important to emphasize that there is a set of tools proposing code automatic generation from UML models, such as AndroMDA [3] and OptimalJ [22]. However, they do not generate a code that reflect exactly the UML2 diagrams design and are focused on specific domain application. In addition, they are not able to checking design rules.

Compared to all these proposals, the advantage of our approach lies in the fact that we *automatically* generate design tests from UML class diagrams to Java code that play the *dual* role of *design test* and *implementation language*. Thus, we check the conformance between the design implementation and the design model, preserving the programmer from the additional task of learning an additional intermediate design test specification language.

5. CONCLUSIONS

In this paper, we introduced a MDA based approach to automatically generate design tests from UML models. In addition, we have built a tool for supporting the proposed approach. This tool was developed itself by pursuing several MDA artifacts, such as metamodels, models and transformations, all them illustrated in the tool's architecture also presented in this paper. We performed several experiments on simple scenarios covering simple designs involving classes, associations, inheritance, attribute and method signatures.

The current implementation of the tool consists of a set of ATL rules that cover UML class, inheritance and association design tests. In order to use our tool, one need only downloading our plugin (on Eclipse) and thus perform the transformations from UML models built in the Eclipse itself or imported from other tools via XMI format.

With the proposed approach and tool, we have been able to generate design tests to check the design of small applications. Both conforming and not conforming implementations have been fully automatically verified against its UML specification using DesignWizard API with no effort to write code based design rules. Despite the simplicity of scenarios, the proposed design tests are very flexible and can be evolved to cope with further details of UML.

UML diagrams, specially class diagrams, are widely used in various software development processes, such as RUP (where UML diagrams are strongly used) or even XP (where UML class diagrams play the role of blueprint diagrams). Therefore, our approach may be used in these processes for checking that the design models are reflected in the code.

As future work, we intend to extend our approach for covering additional design rules concerning other UML class diagram artifacts, such as interfaces. In addition, we also intend to verify behavioral design rules from dynamic UML diagrams, such as interaction diagrams or activity diagrams.

We also intend to investigate the use of OCL expressions as optional notation to express design tests. This would allow the user specifying additional design tests in a platform-independent notation though still remaining with the object-oriented flavor. However, an additional mapping from OCL to Java/JUnit should be provided.

6. ACKNOWLEDGMENTS

This research was supported by grants from CAPES and CNPq of the Brazilian Federal Government.

7. ADDITIONAL AUTHORS

Dalton Serey Guerrero, professor, Department of Computer Science, Federal University of Campina Grande, Campina Grande, Brazil. Email: `dalton@dsc.ufcg.edu.br`.

8. REFERENCES

[1] The peer-review process. *Learned Publishing*, 15:247–258, Oct. 2002.
[2] AMMA Project. Atlas transformation language, 2005. http://www.sciences.univ-nantes.fr/lina/atl/.
[3] Andromda. http://www.andromda.org/.
[4] Atl-dt. http://www.eclipse.org/m2m/atl/.
[5] K. Beck. Embracing change with Extreme Programming. *Computer*, 32:70–77, Oct. 1999.
[6] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems, Nov. 2002. in: Proceedings of Adaptable and extensible component systems.
[7] Design wizard. http://www.designwizard.org.
[8] T. T. Dinh-Trong, N. Kawane, S. Ghosh, R. B. France, and A. A. Andrews. A tool-supported approach to testing UML design models. In *ICECCS*, pages 519–528. IEEE Computer Society, 2005.
[9] D. Evans, J. V. Guttag, J. J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, December 1994.
[10] Find bugs. http://findbugs.sourceforge.net/.
[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
[12] G. Génova, C. R. del Castillo, and J. Lloréns. Mapping UML associations into java code. *Journal of Object Technology*, 2(5):135–162, 2003.
[13] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.
[14] Jas-metamodel. http://www.eclipse.org/gmt/am3/zoos/atlanticZoo.
[15] Javagen and poseidon 3.0. http://www.javagen.com/docs/class-diagram.html.
[16] JUnit. http://www.junit.org.
[17] V. Massol and T. Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.
[18] N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 297–300, Washington, DC, USA, 2006. IEEE Computer Society.
[19] N. Moha, Y.-G. Guéhéneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *ASE*, pages 297–300. IEEE Computer Society, 2006.
[20] Object Management Group. Mof, 2006. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.
[21] Object Manegement Group. Uml 2.0 specification, 2006. http://www.omg.org/technology/documents/formal/uml.htm.
[22] Optimalj. http://www.compuware.com/products/optimalj/.