Checking UML Design Patterns in Java Implementations

Waldemar Pires*, Franklin Ramalho*, Anderson Ledo* and Dalton Serey*
*Universidade Federal de Campina Grande, Campina Grande, Brasil
Email: {neto, franklin, ledo, dalton}@dsc.ufcg.edu.br

Abstract—In a previous work, we presented a technique that allows verifying the conformance between Java implementations and UML class diagrams, using design tests. While it allowed verifying conformance it does so in a sense that does not deal adequately with some design patterns. In such scenarios, there are semantic constraints among the involved elements that UML does not allow our technique to recognize them. Therefore, if one evolves the implementation violating the design pattern, the generated design tests will no longer be capable of detecting the design violation. To address this problem, we propose in this paper an approach based on: (1) UML profiles to explicitly tag UML incorporating design patterns; and (2) a set of design test templates able to recognize the appropriate implementation of these design patterns on Java code. We also present a prototype capable of automatically generating the design tests to verify the design patterns explicated by the UML profile.

Keywords—Design Patterns, Design Tests, MDA, UML Profile;

I. INTRODUCTION

A fundamental activity common to all software processes is the software design. It encompasses a structural description of the software, the data that is to be manipulated by the system, the description of interfaces among system components, and sometimes the algorithms used [1]. A design implements important decisions about the software architecture that play an important role on system development, deployment, and evolution.

Considering this scenario, UML notation is one of the main languages used in system design. UML is widely used to describe both structural and behavioral aspects of system designs.

Design is an important link between the software requirements and its implementation. However, implementations do not always reflect the proposed design. In fact, they rarely do. Obsolete documentation is considered as one of the main causes of low software quality [2]. The conformance checking between design and implementation is still an open issue with few automated tools available.

In a previous work [3], we presented a technique to be used for checking a Java implementation against structural aspects of a design expressed as UML class diagrams. It is able to generate code for design tests that checks whether the implementation code follows the required features expressed in the class diagram. A design test uses static analysis techniques to investigate the implementation and to check the desired properties. The design tests code were automatically generated according to an MDA based approach.

UML is powerful enough to express most design decisions. The elements of UML class diagram covered by our previous approach are able to express these design patterns, however, it cannot be so evident because additional semantic constraints are required. These implicit restrictions among the involved elements may be easily understood by designers and programmers. Still, without an explicit indication of design patterns occurrences at the design, our previous work has no means to guess which restrictions must be considered in order to generate appropriate design tests. As a consequence, it can fail to check the conformance of implementations against designs that involve design patterns.

For instance, consider the Facade design pattern [4]. Using the proposed technique, we can generate design tests to check the presence of all the relationships between external classes and the Facade according to the given UML class diagram. However, such verification does not imply that the pattern was correctly implemented. To do so, we should also verify that there are no relationships between external classes and the classes protected by the Facade. Although this restriction is clear to designers and programmers because it is implied by the Facade design pattern, it is not explicit in the class diagram.

To solve the problem explained before, we propose an approach to extend our previous work in order to support the verification on code of design patterns occurring in UML class diagrams. This extension consists of: i) a set of UML profiles to incorporate the design pattern into UML class diagrams; ii) a set of design test templates to check the correct implementation of each design pattern; and iii) a set of MDA transformations to automatically generate the design tests. The adoption of our approach has a low impact on the software development process. The designer just needs to tag the UML software design with the proposed profiles in the planned design pattern occurrences. From there, a set of executable design tests to verify the correct design pattern implementation in the code is automatically generated.

In this paper we will illustrate our approach by using the Facade design pattern. We have applied our approach, and evaluated it, for others design patterns: Singleton, Proxy, Builder, Abstract Factory, Factory Method, and Prototype. We have chosen structural and creational design patterns because they are suitable to be verified by means of static analysis. Behavioral patterns are not addressed in this work since they would require the execution of the system under
testing. Executing the code would imply reproducing the specific execution environment of the application, which would require an additional complexity to the approach.

The rest of this paper is organized as follows. In section 2 we motivate our work, presenting the problems that emerge when checking an implementation of the Facade design pattern. In section 3, we present and discuss the proposed solution. In section 4, we present a preliminary evaluation of the proposed technique. In section 5, we relate our work to existing approaches concerning design patterns representation and verification on code. Finally, we conclude our paper in section 6.

II. Motivation

To illustrate the problem addressed in this paper, we consider a simple Web system as depicted in Fig. 1. The system was split into three layers: Model, View, and Control, according to the MVC (Model-View-Controller) architecture [5]. MVC is a well-known architectural pattern used in software engineering. Successful use of this pattern isolates business logic from user interface considerations, resulting in an application in which it is easier to modify either the visual appearance of the application or the underlying business rules without affecting each other. For the sake of simplicity, we consider that each layer is represented as a UML package, as shown in Fig. 1.

Assume that a design decision states that the Control layer must communicate with the Model layer only through a Facade. The Facade design pattern claims that there must be a unified interface to a set of interfaces in a subsystem [4]. The designer intention is to express that any class in the class diagram must interact solely through the ModelFacade class (the Facade).

Using our previous [3] approach to check conformance between code and design, we are able to verify on code, for instance: i) the presence of classes with the names given in the UML class diagram; ii) the presence of the method search in the class CommandConsultData; iii) the inheritance relationship between classes CommandSaveData and CommandConsultData and the class AbsCommand; iv) whether class AbsCommand is implemented as an abstract class; and, v) the correct implementation of associations following Gênova et al. [6] guidelines.

Misunderstandings about the application of the Facade pattern can occur in this scenario during the implementation. For instance, consider that the implementation of the class CommandConsultData, in the package Control, accesses directly the class DBHandler in the package Model, as stated in the Code 1. This implementation, thus, breaks the rule of the Facade pattern.

By covering the relationships specified in the UML class diagram, the design tests generated by our previous approach from the design depicted in Fig. 1 indicate that class CommandConsultData is according to the design. In others words, it certificates that all relationships present at the design are reflected on the code. However, the class CommandConsultData has further relationships in the code that are not present in the design. The class CommandConsultData uses the class DBHandler directly (line 4), and such relationship breaks the Facade design.

```
class CommandConsultData extends AbsCommand{  
    public String search(String query){
        ...  
        BDHandler bd = new BDHandler();  
        bd.openConnection();  
        return bd.makeQuery(query);  
    }  
    ...  
}
```

This incomplete verification occurs because in our previous approach we check only the presence of relationships specified at the design. It does not verify the relationship absence because: i) it allows a no rigid conformance checking: more freedom is given to developers to implement the system, and, thus, they can create additional attributes, methods and auxiliary classes; ii) it can be inefficient: to check the absence of others relationships, we must verify the relationship presence or absence between all model classes (that has N*N complexity in worst case, where N is the number of classes in the model).

Almost all the design patterns have implicit semantic constraints that make them difficult or impossible to check using our previous approach, due to the same reasons aforementioned. It is certain that these limitations are valid for all design patterns in which implicit restrictions on the possible relationships exist.

III. Our Approach

By considering the scenario illustrated in the previous section, we propose an extension of our previous approach to create a library of design patterns verifiable on Java source codes. This approach consists of: i) a set of UML profiles to incorporate the design pattern in UML class diagrams; ii) a set of design test templates to verify the correct implementation of the design pattern on Java source codes; and iii) a MDA approach to generate automatically these design tests.

The Fig. 2 depicts the adoption of our approach in a generic software development process. In the step 1, the software designer applies our profiles to the UML class diagram of the system and put the class diagram XMI as input of our tool. In the step 2, our tool, based on the design test templates, generates the design tests to verify each design patterns occurrence. Finally, in the step 3, the design tests generated are executed with the system code as input. As result the tests point the classes that are not in conformance with the design pattern.

MDA is a software engineering approach defined by the OMG. It represents just one view of MDD (Model-Driven Development) process of the software development.
The structural design of a web system specified through an UML class diagram.

Figure 1.

Steps to adoption of our approach.

Figure 2.

Development), though it is perhaps the most prevalent at present [7]. 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 the ontology organization 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 models before mentioned. In this sense, there are a set of transformation languages, such as QVT [8] or ATL [9], supported by their respective transformation engines responsible for executing them.

We pursued MDA as the approach to generate design tests because: i) we consider that the software design is specified using UML (PIMs); and ii) Java code is, in fact, a PSM. Thus, by reusing MDA standards and frameworks and additionally by specifying the transformations from i to ii, we take profit from the MDA benefits. Our MDA-based approach is illustrated in Fig. 3.

At the left side of the Fig. 3 are the input models, i.e., the system design specified by the UML class diagrams. The design pattern profile is applied to these diagrams to explicit the design pattern application. Both, UML class diagrams and the design pattern profile, must follow the UML metamodel provided by the OMG consortium [10]. At the right side are the output models, i.e., the design tests that must be in conformance with the Java Abstract Syntax (JAS) [11] metamodel. The design tests are specified as JUnit tests, and they reuse the framework Design Wizard [12] [13] to extract the code structure. In the center there are the ATL transformations that automatically generate the design tests to verify the implementation of the design patterns. For each design pattern there is an ATL matched rule producing the design test able to verify it. ATL supports a set of formats, including the standards proposed by the OMG.

In addition, We specify an ATL transformation rule to each line pertaining to the design test to be generated. Therefore, we intend to make easier the design test template extension, since adding, removing or modifying any code line in the design test template requires to change only a specific and individual ATL called rule that is responsible for generating this line.

Due to our MDA approach is based on ATL transformations, the generation of a specific design test can be extended or completely changed, without interfering in the generation of others design tests. In addition, to add a new design test to be verified in our prototype, it is necessary to (i) define an UML profile covering the elements to be added to the UML model in order to identify the design pattern application; (ii) create a design test template for the new design pattern;
and (iii) create a new ATL matched rule to automatically identify the design pattern occurrence in the UML model as well as their respective ATL called rules responsible for generating each code line of the new design test responsible for verifying the Java implementation of that design pattern.

In the following subsections we explain our approach in more details. In particular, we illustrate its application to the Facade design pattern. The Facade design pattern claims that must exist an unified interface to an entity set in a subsystem [4]. Here, we consider that the correctly implementation of this is when none external class of the Model package should not invoke directly any internal class, except the Facade class. This implementation is relatively restrictive, but we adopted it to explain whole approach. In subsection 3-A, we describe the profile to explicitly tag the Facades in the UML class diagrams. In subsection 3-B, we present the design test template to verify the correct Facade implementation in Java source code. Finally, in subsection 3-C we show the executable ATL transformations that are able to automatically generate the design test for a given design.

### A. UML Profile

Profiles are based on additional stereotypes and tagged values that are applied to UML elements, such as classes, attributes, methods, associations and so on. When a profile is branded to a UML model, the semantic meanings of the stereotypes are implicitly attached to the elements of the model.

Considering the Facade design pattern, we propose the profile depicted in Fig. 4. It contains the Facade stereotype that must be applied to UML classes, as specified in the profile. The class extended by this stereotype is that playing the role of the Facade. In order to preserve the semantic constraints inherent to the Facade class, we have enriched the Facade profile with the OCL constraint shown in Code 2. This constraint assures that all classes that do not belong to the package protected by the Facade cannot have any kind of relationship with any class protected by the Facade.

![UML profile for the Facade design pattern.](image)

To facilitate the comprehension of this OCL constraint we created several local variables: (1) `packageClasses`, a set with all classes belonging to the package protected by the Facade; (2) `protectedClasses`, the set `packageClasses`, but the Facade; (3) `packageRelationships`, a set with all relationships belonging to the Facade package; (4) `internalRelationships`, a set with all relationships where all member ends are classes belonging to the package where the Facade is contained; (5) `outgoingRelats`, a set with all directed relationships outgoing from classes protected by the Facade; and (6) `othersRelats`, a set with all relationships of the Facade package excluding the permitted relationships enfolding protected classes (`internalRelationships, outgoingRelats`).

The invariant states that the set `othersRelats` does not have to contain any relationship enfolding the classes protected by the Facade, i.e. it must contain just relationships with the Facade class.

```java
context Facade inv:
    let facade: Class = self.metaClass->first()
    in let packageClasses: Set(Class) =
        class.package.ownedType->select(s | s.oclsTypeOf(facade))
    in let protectedClasses: Set(Class) =
        packageClasses->excluding(facade)
    in let packageRelationships:
        Set(0 relation) =
            class.package.ownedType->select(r | r.oclsKindOf(UML2.Relationship))
        in let internalRelationships:
            Set(0 relation) =
                packageRelationships->select(s | s.relateElement->forAll
                (r | packageClasses->includes(r)))
        in let outgoingRelats:
            Set(UML2.Relationship) =
                packageRelationships->excluding(outgoingRelats)
        in let othersRelats: Set(UML2.Relationship) =
            packageRelationships->excluding(0 relation) 
            ->excluding(0 relation) 
            ->forAll(r | not protectedClasses
            ->includes(r));
```

### B. Code Verification

The creation of an UML profile is not enough to accomplish our goal of verifying design patterns on Java source code. To address this problem we propose a set of design tests to verify the correct implementation of the design pattern on a low abstraction level, i.e. on the source Java code. Design tests look like the unit test, however instead of verifying if a code has an expect behavior, it verifies if an implementation is in conformance with a desired structural design. We have reused two technologies for specifying the design tests: i) the Design Wizard, a library capable of extracting from the Java code its structure; and ii) the JUnit Framework, a well-known test framework able to test the structure extracted by the Design Wizard.

For instance, the Code 3 is an excerpt of the design test template to verify the correct implementation of the Facade design pattern.

Line 3 gets a `ClassNode` instance representing the Facade class. The tag `<FacadeClass>` must be replaced by the name of the class that plays the role of the Facade. If the
class does not exist an exception `InexistentEntityException` will be thrown, and the test will fail alerting about the unexistent class. Line 5 creates a set with all classes (`packageClasses`) belonging to the package in which the Facade is contained. Lines 6-9 create the same set `packageClasses`, but without the Facade class (`systemClasses`). Lines 10-11 create a set with all classes that are not contained in the package in which the Facade is specified (`extClasses`). Lines 12-18 represent an iterative statement (Java `foreach`) that scans the set `extClasses`, verifying if some of these classes accesses some class protected by a Facade, `systemClasses`. If some of these verifications is true, i.e., any external class makes reference to a protected class, it means that the design pattern Facade has not been respected.

**C. Use Example**

Initially, consider as use example the Web system aforementioned in the motivation section (section II). According to a strategic decision in this system, any external class of the Model package should not invoke directly any internal class, except the Facade class ModelFacade. Therefore, in this section, we show how to pursue our approach to verify and maintain the correct Facade design pattern implementation as well as the system design.

The first step is to adopt our approach and brand it in the pattern occurrence. Therefore, in the web system example, we could extend the Web System class diagram by applying the Facade profile (Fig. 4). The resulting UML class diagram is shown in Fig. 5. Thus, we can explicitly represent the application of the design pattern pattern Facade. Thereby, we are able to verify on system design that the Model layer only must be accessed through the class ModelFacade (the Facade class). And, we could check Facade profile invariant (and any other OCL invariant) using an appropriate tool like MDT-OCL. Indeed, the profile Facade invariant is respected in the Web system class diagram. However, when the Web system evolves, new classes and relationships could be added to the class diagram design, breaking the design pattern. And, with the explicit design pattern application in the class diagram and checking the design pattern invariants, we could avoid misunderstanding and bad implementation of design pattern.

The second step is to generate the respective design tests to verify each design pattern occurrence. Our prototype is able to collect the design patterns occurrence in the class diagram and populate the design test template with this information. The prototype uses a MDA approach to identify a proposed design pattern profile occurrence and generate the design tests to verify its correct implementation in the code. The generated test to verify the Web system Facade is shown in the Code 4.

The last step is to execute the Code 4 to test the Web system code. As shown before, the class CommandConsultData (Code 1) is respecting the class diagram design; however, it is not respecting the Facade design pattern. Therefore, executing the tests generated by our previous approach no errors are detected. However, the design pattern tests indicate that the class CommandConsultData is not respecting the Facade design pattern. The design tests execution result is depicted in Fig. 6, where we can see that the JUnit Framework test fails, showing that the class CommandConsultData is not respecting the Facade of Package Model (accessing the class DBHandler).

### Code 3

**Excerpt of design test template to verify the Facade design pattern.**

```java
... public void testFacade(){
  ClassNode c = dw.getClass("FacadeClass");
  PackageNode p = c.getPackage();
  Set<ClassNode> packageClasses = p.getClasses();
  Set<ClassNode> systemClasses =
    new HashSet<ClassNode>();
  systemClasses.addAll(packageClasses);
  systemClasses.remove(c);
  Set<ClassNode> extClasses = dw.getClasses();
  extClasses.removeAll(packageClasses);
  for (ClassNode classNode : extClasses) {
    Set<ClassNode> users =
      classNode.getUsers();
    if (classNode.getUser() == null) {
      this.assertFalse(systemClasses.contains(classNode2));
    } else {
      this.assertFalse(systemClasses.contains(classNode2));
    }
  }
  ...}
```

### Code 4

**Design to verify the correct Facade implementation.**

```java
... public void testFacade_ModelFacade(){
  ClassNode c = dw.getClass("ModelFacade");
  PackageNode p = p.getClass();
  PackageNode systemClasses =
    new HashSet<ClassNode>();
  systemClasses.addAll(packageClasses);
  systemClasses.remove(c);
  Set<ClassNode> extClasses = dw.getClasses();
  extClasses.removeAll(packageClasses);
  for (ClassNode classNode : extClasses) {
    Set<ClassNode> users =
      classNode.getUsers();
    if (classNode.getUser() == null) {
      this.assertFalse(systemClasses.contains(classNode2));
    } else {
      this.assertFalse(systemClasses.contains(classNode2));
    }
  }
  ...}
```

### D. Others Design Patterns

Our approach is capable to verify others design patterns: Singleton, Proxy, Builder, Abstract Factory, Factory Method, and Prototype [4]. In this section, we show shortly how our approach treats them.

**Singleton.** We created a profile with a stereotype, named Singleton, applied to the metaclass Class. Additionally, we brand this stereotype with two OCL constraints: the first assures that the visibility of every constructor method must be private or protected; the second asserts that there must be at least one static method that returns an object with the same type of the class tagged with the stereotype. The design test to verify this pattern on source code is very similar to its OCL constraint. It verifies whether the visibility of every constructor method (pertaining to the Singleton classes) is either private or protected. This verification is necessary...
Facade profile applied to the class diagram of the Web System shown in Fig. 1.

Figure 5

Results of the Facade design test execution for the Web System.

Figure 6

because the implementation can have additional constructor methods that could break the Singleton pattern. To verify the existence of a static method is not required because it is assured on the model layer by the OCL constraints. Our previous approach [3] was already able to verify the existence of this method and if it was correctly implemented.

Proxy. We created a profile containing a stereotype hierarchy due to the existence of many Proxy variations. Firstly, we defined an abstract stereotype, named GeneralProxy, applied to the metaclass Class. Therefore, each Proxy variation must specialize GeneralProxy. For instance, to deal with the virtual Proxy [4] we created a stereotype named VirtualProxy. As part of the virtual Profile we also defined a stereotype named DelegateMethod, to be applied to the metaclass Operation, representing the method which controls access to the class protected by the Proxy. An OCL constraint was branded to assure that the class tagged with the stereotype VirtualProxy must have at least one method tagged with the stereotype DelegateMethod. Furthermore, the design test to verify this design pattern must verify whether the method tagged with DelegateMethod calls the method with the same name of the class protected by the Proxy.

Builder. The profile to this design pattern contains three stereotypes, applied to the metaclass Class, each one for a design pattern role: Director, Builder, and Product. Also, we defined an OCL invariant to assure the relationship between the classes. For instance, the class tagged with the stereotype Director must have (at least) one association with the class tagged with Builder. And, the design test verifies (1) if only the ConcreteBuilders (Builder sub-classes) could instantiate the Products, and (2) if only the other design pattern participants (Directors and Product) can invoke methods of the Product.

Abstract Factory. For this design pattern we created a profile with two stereotypes, applied to the metaclass Class, each one for a design pattern role: Factory, and Product. The OCL invariants assure that the external classes (Clients) are only linked with the Products (never with the concrete products). Furthermore, the design tests verify if only the concrete factories (Factory subclasses) instantiate the concrete products (Product subclasses).

FactoryMethod. We created a profile with two stereotypes, named Product and Creator, applied to the metaclass Class. Also, we defined an OCL invariant to assure the relationship between the ConcreteProduct (Product subclass) and a ConcreteCreator (Creator subclass). Furthermore, we defined the design test template able to verify if only the ConcreteCreator (Creator subclasses) instantiates themselves.

Prototype. We created a profile with a stereotype, named Prototype, applied to the metaclass Class. Furthermore, the design tests verify if only a ConcretePrototype (Prototype subclasses) could create an instance of themselves.
For each one of these design patterns there is an ATL transformation set to apply respective design test template for each occurrence in the UML software design.

IV. Evaluation

In this section, we present a preliminary evaluation of the proposed technique. We evaluated its feasibility, by designing and implementing a prototype that supports the technique, as well as the templates for seven design patterns, viz. Facade, Singleton, Proxy, Builder, Abstract Factory, Factory Method, and Prototype. To evaluate the effectiveness of the technique to check actual Java implementations, we conducted experiments in which both correct and incorrect implementations were automatically checked by our tool against given UML class diagrams. Finally, we evaluated the performance of our tool, by profiling the time required for both generating and executing design tests.

Appropriate subjects to our experiments consist of a software project implemented in Java with available class diagram artifacts and with at least one design pattern of our interest. Due to the difficulty to find real, well known open source projects that satisfy all these requirements, we decided to identify a Java open source project and generate their class diagrams, using reverse engineering tools. This strategy allowed us to run our experiments in real, well known projects. Thus, the steps we followed to prepare the subjects were: (1) identifying and obtaining candidate Java projects; (2) generating their class diagrams, using reverse engineering tools; (3) extending the class diagrams to apply the proposed profiles and tags to indicate the presence of design patterns.

We adopted some well-known Java open source projects to use as subjects in our experiments. In a preliminary evaluation, we used two Java open source projects as subject of our experiments: Findbugs, a tool to inspect software for potential bugs; and Apache Ant, a well known tool to automate software build processes. However, this evaluation could be considered limited, because just three design patterns (Facade, Singleton, and Proxy) was covered. Therefore, we adopted five other projects that implemented the remaining design patterns: JHotDraw v5.1, a framework to manipulate two-dimensional graphs for a structured drawing editor; JRefactory v2.6.34, a tool capable to perform action to refactory java codes; JUnit v3.7, the famous unit testing framework for the Java programming language; MapperXML v1.9.7, a framework for web applications, based on the architectural pattern MVC; QuickUML 2001, a tool to design UML models. The same projects were used as a case study of the Gueneuneuc work [15]. These projects and their design pattern implementations are available in the P-MARt database [16].

To produce the class diagrams, we used a trial version of Omondo [17], a visual modeling tool that conforms to the UML2 metamodel defined by the OMG. To apply the profiles and tags, we manually inspected the code, the documentation and the diagrams produced in the previous step. Once we identified the design patterns in the project (by P-MARt database or analyzing the source code), we manually applied the profiles to the class diagrams to indicate the presence of design patterns.

The results we collected for this first analysis are summarized in Table 1. Each line of the table refers to one of the seven projects studied. Each column presents results to one of the design patterns verified. In each cell, we can see how many correct implementations were checked by our tool and how many we expected. For instance, our technique checked 6 correct implementations of the Facade design pattern in project Findbugs, while 7 should have been confirmed. For the Facade pattern these results indicate that the technique considered as correct implementation 83% of all occurrences identified for this pattern. For the Singleton, the technique considered 92% of all correct occurrences. For the Abstract Factory, the technique considered 50% of the identified occurrences as correct implementation, worst result. For the Proxy and Builder pattern, the technique was able to correctly check all implementations. However, it must be mentioned that the number of occurrences is rather small - only 2 proxies were identified for each pattern. For the Prototype, our technique also was able to confirm all correct implementations. However, in our worst case, only one occurrence was identified. Finally, we consider as the better results for the Factory Method, where the technique was able to correctly check all five correct implementations.

In the preliminary experiments, we also conducted a second experiment in which we tagged design patterns in entities that did not use them. We randomly chose elements of the class diagrams that were not part of the previously detected design patterns and tagged them along. For each project, we tagged 30 incorrect design patterns - 10 of each pattern under study. Thus, in total, we tagged 60 incorrect design patterns. As we expected, the technique and the tool were able to indicate, in 100% percent of the cases, that the patterns were not considered as correctly implemented.

The two cases in which the technique did not work well deserve special attention. Analyzing the Findbugs project, we identified that there is one Facade implementation that actually does not strictly satisfy the requirements for the Facade design pattern. The mentioned class is edu.umd.cs.findbugs.ba.Hierarchy, which is documented in the Findbugs API as a Facade. Our tool correctly pointed out that class edu.umd.cs.findbugs.Lookup directly uses class edu.umd.cs.findbugs.ba.XClass without accessing it through the Facade (class Hierarchy, mentioned before, maybe indicating a reference leak). If the strict interpretation of the Facade pattern is to be used, then the fact pointed by the tool is a design flaw. Another design flaw possibly reported by our technique is that the class edu.umd.cs.findbugs.config.UserPreferences does not strictly satisfy the requirements for the Singleton pattern, as the
API claims it to be. In fact, the constructor of the class is public, which goes against the classical implementation of the Singleton pattern. Of course, these design flaws can be specific points in which the designer deliberately wants to break the rules. Whatever the reason, we must enforce that the technique presented has been able to detect and report these cases by automatically checking the code, and its use would allow designers to be aware of these facts.

Our final analysis was about performance. We measured the time our technique required to generate and execute the complete test suite to verify the three design patterns (Facade, Singleton, and Proxy) considered in the experiments for three projects with different sizes: Findbugs, JUnit, and the Apache Ant. The results are summarized in Table 2. As expected, the time is proportional to the number of classes in the project. However, further analysis is required to understand why this relation does not seem to be linear. Finally, we must mention that execution time was absolutely within our expectations. As it can be seen, even for the larger system, actual times are below 20 seconds, which is considered acceptable for an automated test suite.

Further analysis is still necessary to see how these results generalize. However, this preliminary evaluation has been enough to convince us that the idea is minimally feasible, effective and efficient and, thus, that further effort should be put in providing more templates for other patterns and we should proceed to a more rigorous evaluation.

V. RELATED WORK

Considering the design pattern representation in UML diagrams, Dong et al. [18] proposed a UML extension to make design patterns explicit in class diagrams. They extended UML through a set of UML profiles, each one specifying a design pattern. However, only a small set of design patterns is covered by their work. For instance, they do not cover the Facade design pattern. Moreover, the approach is not considered as a support to the verification of the conformance of implementations and the corresponding designs.

There are some works that specify design patterns using formal specifications. For instance, Eden [19] proposes the formal language LePub to specify patterns. However, to our best knowledge, there is no available tool that allows verifying these properties neither on models nor on code.

Dong et al. [20] proposed an approach to guarantee that the structural constraints related to a design pattern will be respected when a system design evolves. To achieve it, they defined a process to evolve design patterns in terms of two transformation levels: the first describes the basic transformations (e.g., addition or removal of modeling elements, such as classes or relationships); and the second describes a process to evolve each design pattern based on the first level transformations. Following this evolution process, they can assure automatically that the design pattern is not violated on the models. This work assures the correct implementation and evolving of design pattern on UML class diagrams, i.e. on model level. Again, this work focuses on models, and nothing is discussed about the implementation code.

The PTIDEJ project (Pattern Trace Identification, Detection, and Enhancement in Java) [21] aims at developing a tool suite to evaluate and to enhance the quality of object-oriented programs. The proposed tool creates a model of a program from its source code by static and dynamic analysis. By investigating the model, the tool is able to identify design patterns and design defects. However, the proposed tool does not support customization. Additionally, this tool does not use the standards proposed by the OMG to describe their models. Its generated models are expressed as PADL models.

Izurieta et al. [22] evaluate the testability consequences of the so called grime buildup in object oriented design patterns. Their approach uses the reverse engineering to build the UML diagrams from code [23]. And, by using the approach [24], they verify the correct implementation of design patterns on the generated UML diagrams. As a final step, they evaluate the pattern evolution on the code by effort to test it using the Binds approach [25]. This work differs from ours because our work considers a previous existence of a system design and that the code must reflect this design respecting the design pattern constraints. Their
approach verifies the design pattern on models produced by reverse engineering process, and this process is information lost prone. In this paper, we just use reverse engineering to evaluate our approach, because we did not find out a software project implemented in Java that provides its class diagram artifacts. Additionally, although this approach automates the design pattern recognition, the identification of which relationships participate of the design pattern is a manual process. Conversely, our approach is completely automatic.

Guéhéneuc et al. [15] propose an approach to semiautomatically identify microarchitectures that are similar to design motifs in source code and to ensure the traceability of these microarchitectures between the design and the implementation. Design motifs means the same that we call as design pattern implementation, in other words, how the design pattern was implemented in the code. This work uses a reverse engineering technique, based-on dynamic and static analyses, to generate the system models. These models are based-on metamodel PADL (Pattern and Abstract-level Description Language), a UML-like metamodel. This approach is able to discover pattern occurrence and where this pattern could be badly implemented. Our approach does not generate the design from code (we use it to evaluate our approach). We consider that the design must be created before the code, and that the design pattern projected in the class diagram must well expressed and well implemented in the code. Furthermore, our approach adopts the OMG standards and the DEMIMA uses proprietary standards.

France et al. [26], [24] proposed a technique to specify patterns in UML. The technique consists of UML metamodel specialization incorporating a Structural Pattern Specialization (SBS) and an Interaction Pattern Specialization (IPS) to describe the structure and behavior of the pattern, respectively. They also propose a tool that verifies if patterns expressed by their UML metamodel specialization are followed by UML models. However, they focus on verifying the design patterns on models. None of the cited works addresses the problem of conformance checking between code and models.

VI. Conclusions

In a previous work, we presented a technique that allows verifying the conformance between the structural design specified by UML class diagrams and its Java source code. In this paper, we proposed and discussed an extension for this approach allowing to verify models annotated with design patterns. It has been achieved by: i) incorporating design patterns in UML class diagrams through UML profiles; and ii) verifying them on code through design tests automatically generated according to an MDA-based approach.

We implemented and evaluated our approach to the following design patterns: Facade (illustrated), Singleton, Proxy, Builder, Factory Method, Abstract Factory, and Prototype. In addition, some design patterns can be verified strictly by design tests generated by our previous approach, such as the Composite design pattern that is focused on the usage of inheritance and associations to structure classes aiming to treat uniformly many related objects, and multiple objects can be treated as one [4]. Therefore, using our previous approach to check if the associations and inheritance relationships were implemented correctly is enough to verify the correct structural implementation of this design pattern.

We evaluated the effectiveness of our technique by checking actual Java implementations. We conducted experiments in which both correct and incorrect implementations were automatically checked by our tool against given UML class diagrams. Finally, we evaluated the performance of our tool, by profiling the time to both generate and execute design tests. Further analysis is still necessary to see how the found results generalize. However, this preliminary evaluation has been enough to convince us that the idea is minimally feasible, effective and efficient and, thus, that further effort should be put in providing more templates for other patterns and we should proceed to a more rigorous evaluation.

As we said, OCL is used to define constraints on the design patterns for the class diagram models, and handwritten templates are used to generate Java code to check these constraints in the Java codes. To maintain these two artifacts consistent and complete can be a hard task. Therefore, we are developing a technique to use OCL-to-code transformation to generate the checking code.

We consider as limitations of this work: a poor User Interface, this work has not a friendly UI, all interactions with the tool must be done by command line prompt to provide the input model location and the output location; the transformations are in hard code, whole transformations code were implemented as a well structured ATL transformation set that was developed to make easy extend any transformation, but M2M (Model to Model) Transformations are too much verbose to deal with detailed code; strict design pattern verification, if only one assert fails, our approach considers as whole design pattern was not correctly implemented; not complete, some cases our approach can declare false positives, for instance, some classes can be accessed directly by class reflection or very late bind, and it is not detected by our approach (we verify just the structural implementation).

As future work, we intend: to improve the UI, developing a Graphical User Interface, as Swing or GWT; to extend the approach by using a code template transformation as JET [27] or MOFScript [28] instead of using ATL transformations; to soften our strict design pattern verification by showing the percentage of how the pattern is well implemented, or even suggesting corrections instead of showing up JUnit fails. Others improvements are to implement and to evaluate others structural and creational design patterns, like Bridge and Abstract Factory [4]. Additionally, we also intend to cover additional UML elements like components (considering UML Component Diagram). Thereby, our approach
could cover more complex architectural patterns, like MVC [5]. In addition, we also intend to verify behavioral design rules from dynamic UML diagrams, such as interaction or activity diagrams. We can make this verification by monitoring the code. Therefore, our approach could also cover additional behavior design patterns.

ACKNOWLEDGEMENTS

This work was supported by grants from CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), FAPESQ/PB (Fundo de Apoio à Pesquisa do Estado da Paraíba) and PET (Programa de Educação Tutorial).

REFERENCES


