Structural Conformance Checking with Design Tests:
An evaluation of usability and scalability

João Brunet, Dalton Serey and Jorge Figueiredo
Department of Computer Science / DSC
Software Practices Lab / SPLab
Campina Grande, Paraíba, Brazil.
{jarthur,dalton,abrantes}@dsc.ufcg.edu.br

Abstract—Verifying whether a software meets its functional requirements plays an important role in software development. However, this activity is necessary, but not sufficient to assure software quality. It is also important to check whether the code meets its design specification. Although there exists substantial tool support to assure that a software does what it is supposed to do, verifying whether it conforms to its design remains as an almost completely manual activity. In a previous work, we proposed design tests — test-like programs that automatically check implementations against design rules. Design test is an application of the concept of test to design conformance checking. To support design tests for Java projects, we developed DesignWizard, an API that allows developers to write and execute design tests using the popular JUnit testing framework. In this work, we present a study on the usability and scalability of DesignWizard to support structural conformance checking through design tests. We conducted a qualitative usability evaluation of DesignWizard using the Think Aloud Protocol for APIs. In the experiment, we challenged eleven developers to compose design tests for an open-source software project. We observed that the API meets most developers’ expectations and that they had no difficulties to code design rules as design tests. To assess its scalability, we evaluated DesignWizard’s use of CPU time and memory consumption. The study indicates that both are linear functions of the size of software under verification.

Keywords—design verification; structural conformance checking; design test.

I. INTRODUCTION

Structural conformance checking refers to evaluating whether an implementation meets its design. While many aspects of the design may be irrelevant to check, some critical decisions are crucial to ensure software quality. In particular, it is known that some design decisions are key aspects for software evolution [1].

A common approach to perform conformance checking is to specify design rules and check the implementation against these rules. A design rule is a constraint that a designer expects to be satisfied by developers while coding and evolving an implementation. Using design rules, a designer may specify not only what the program must do, but also how it must look like internally — e.g. she might specify the components of a software and how they should be organized.

There have been several academic attempts to automate structural conformance checking [2], [3], [4], [5]. The state-of-the-practice, however, consists in manually inspecting the code [6], [7], which is error-prone and does not scale as software grows.

State-of-the-art approaches still fall short of practical needs. Many approaches use formal languages to express design rules. However, specifying rules as formulas has no appeal to developers (in fact, it discourages them). While more operational languages exist, they usually require developers to learn additional syntax, semantics and concepts to write design rules. It is a well known fact, that despite its academic success, Architecture Description Languages have not been widely adopted in practice [8], [9]. ADLs require learning their syntax and concepts to deal with design rules. Generally, this compromises the adoption of these approaches, once it increases their learning curve.

On the other hand, testing is the de facto verification method in industry. Among other attractive qualities, tests are easy to understand, to write and to maintain. Automated tests, in particular, can be used as an executable complement to specifications that help clients and providers agree on results delivered. Several aspects of software products can be checked using tests. However, tests are always used to check whether a software performs as it is expected to do. Checking internal properties, however, is not in the realm of software testing. No work, to the best of our knowledge, has been reported on using tests to check whether the implemented software complies to given design rules — i.e. whether it has been implemented as it was supposed to be.

In a previous work [10], we introduced a preliminar idea to check conformance through tests — the concept of design tests. Design tests are an application of the concept of tests to design conformance checking. A design test is a test-like program that checks whether an implementation complies with a given design rule. While conventional automated tests check whether the software produces an expected answer to a given stimulus, design tests check whether software components exist in the implementation and how they are interconnected. Therefore, while conventional tests increase confidence that the software does what it is supposed to do,
design tests increase confidence that the software has been built as expected.

In this paper we present a complete description of our approach and address two major important aspects aroused from our previous work: the usability and scalability of design test tool support.

To support the use of design tests in practice, we developed DesignWizard. It is a fully fledged API that allows writing design tests for Java projects. DesignWizard has been developed to be used in conjunction with the popular JUnit testing framework [11] (the de facto standard for unit testing in Java). We have performed a study on the usability of the DesignWizard API as a means to write design tests for Java. We have also evaluated how DesignWizard scales as software under testing grows. The qualitative usability evaluation was conducted using the Think Aloud Protocol for APIs [12], [13]. This protocol is an adaption of the Think Aloud Protocol [14] - the standard mechanism to gather data in usability testing. It demands users’ verbalization as they perform pre-defined tasks in the system being analyzed. The protocol was adapted due to the fact that an API has different aspects to be analyzed from that of software systems.

In the experiment, we challenged eleven developers to compose design tests for an open-source software project. We observed that the API meets most developers’ expectations and that they had no difficulties to code design rules as design tests. To assess its scalability, we evaluated DesignWizard’s use of CPU time and memory consumption. The study indicates that both are linear functions of the size of software under verification, and that constants involved make it a very attractive solution.

The rest of this paper is structured as follows. In Section II, we describe the design testing approach to check conformance between implementation and design rules. Section III describes how to compose a design test for Java projects using the DesignWizard API. Later, in Section IV and Section V, we report the evaluation and results of our approach. In Section VI, we discuss related work. And in Section VII, we conclude the paper, presenting our final remarks and future work.

II. STRUCTURAL CONFORMANCE CHECKING WITH DESIGN TESTS

Typically, most software projects define design rules to control their structure. These rules are used as contracts that must be followed by developers when the software is under construction.

In order to perform structural conformance checking by means of design rules, it is necessary to specify them and verify the implementation against these rules. Regarding the specification, a design rule can be expressed as a first order logic formula under a domain expressed by a design metamodel. For example:

Every class inside command package must extend Command class.

In terms of first order logic, this rule is specified as follows:

\[ \forall c \text{ contains}(command, c) \Rightarrow \text{extends}(c, \text{Command}) \]

However, instead of requiring designers to learn a formal or a declarative language to specify design rules, we developed an approach, named design test, in which the key concern is to write design constraints in the same programming language as that of the software under verification. This is achieve through the use of two components:

- A structure analyzer API that allows developers to inspect the structure of the code;
- A testing framework that provides assertion routines.

Figure 1. Overview of checking code with design test.

Figure 1 illustrates an overview of the process that we have idealized to check code against design rule with Design Tests. In the first phase, designer idealize a specific design rule that it is important to be followed by developers during software evolution. After that, in the second phase, the designer uses a structure analyzer API and a testing framework to specify a design test that programmatically describes this rule. Finally, the test is executed using the infrastructure provided by testing framework. As can be seen in Figure 1, the result of the execution is reported as a success or failure, which is the mechanism that the testing framework uses to report functional tests results. However, the result of a design test execution has a different semantic from the results of a functional one. A success in the execution of a design test means that the code analysed follows the design rules described by the test. By the other side, a failure denotes that the code is not in conformance with the specified design rules.

Design tests can describe a number of structural properties as design rules. A common example of design rule is to
restrict the communication between unrelated components of the software to avoid unnecessary coupling between them. A classic instance of this constraint establishes that presentation components must not directly access data components. Pseudocode 1 implements this design rule as a test.

Pseudocode 1 Design test pseudocode.

```
1 presentation = getPackage("presentation")
2 data = getPackage("data")
3 calledPackages = presentation.getCalledPackages()
4 assertFalse(calledPackages.contains(data))
```

The main idea is to obtain information about the structure of the code and write assertions routines that check whether this structure satisfies the specified properties. In this example, the test acquires the set of packages accessed by the presentation package and verify whether this set does not contain the package data.

For the sake of simplicity, let’s suppose that we want to check whether the Electronic Calculator implemented by the Pseudocode 2 is in conformance with the design test specified by the Pseudocode 1.

Pseudocode 2 Electronic Calculator Example.

```
1 package presentation;
2 class GUI {
3     sum() {
4         Data.setSum(value);
5         Main.sum();
6     }
7 }
8 package data;
9 class Data {
10     setSum(int) {
11         ...
12     }
13 }
14 package main;
15 class Main {
16     sum() {
17         Data.setSum(value)
18     }
19 }
```

In order to do so, we only have to execute the design test to automatically perform the structural verification. To better explain the concept of a design test and how it works, it is interesting to give a practical example. Table I illustrates the step-by-step execution of the test for the Electronic Calculator verification.

In the assertion phase (line 4) the test checks whether the calledPackages set does not contain the data package. At this moment, when executed, the test reveals a failure, once the calledPackages contains both, main and data package. This scenario happens because the method presentation.GUI.sum() calls the method data.Data.setSum() (line 4).

<table>
<thead>
<tr>
<th>Executed Line</th>
<th>presentation</th>
<th>data</th>
<th>calledPackages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1</td>
<td>presentation</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Line 2</td>
<td>presentation</td>
<td>data</td>
<td>null</td>
</tr>
<tr>
<td>Line 3</td>
<td>presentation</td>
<td>data</td>
<td>{data,main}</td>
</tr>
</tbody>
</table>

Figure 2 is a graphical representation of the Electronic Calculator implemented by Pseudocode 2. The highlighted part indicates the undesirable relationship between presentation and data packages that the execution of design test reveals.

Due to the fact that a design rule is a first order logic formula, it may contain quantifiers, logic operators and predicates under a domain expressed by a design metamodel. Once design tests are programs, the own programming language provides the quantifiers and logic operators necessary to write a design rule as a design test. Additionally, predicates may be constructed with the aid of a structure analyzer API that provides information about the entities of the code and their relationships. This API must contains services like the following:

- `getPackage("package name") : Package`
- `getClass("class name") : Class`
- `getMethod("method signature") : Method`
- `getField("field name") : Field`

These methods are necessary to get access to the entities of the code under verification. With an object that represents an specific package, class, method or field, the developer can query about the relationships of this component. For example, in order to obtain the classes that are inside a package, the developer can use the method `getClasses()` of the Package abstraction.

For the sake of clarity, let’s get back to the example explored before and try to specify it as a design test:

\[ \forall c \text{ contains}(\text{command}, c) \Rightarrow \text{extends}(c, \text{Command}) \]

As we can see, this rule is based on the universal quantifier (\(\forall\)). Observing Pseudocode 3, which is the design test implementation for the rule, we can see that design tests use the own programming language support to implement the universal quantifier (FOR command). Besides that, the
information about the classes inside command package is obtained using the getClasses() method provided by an API. At last but not least, an assertion routine is used to check whether each class of command package extends the Command class.

**Pseudocode 3** Design Test Example.
```java
1 command = getPackage("command")
2 classes = command.getClasses()
3 FOR class in classes DO
4   assertTrue class.extends(getClass("Command"))
5 DONE
```

So, to write a design test it is not necessary to learn a different language as that of the software been checked. For example, every programming language can provide iteration over a data structure, which may be used as an alternative to the universal quantifier. The same happens to the existential quantifier and the logic operators. For instance, the Pseudocode 4 is the design test implementation for the following design rule:

There must exist a method called execute() or a method called wait() in Command class.

**Pseudocode 4** Design Test example for existential quantifiers and logic operators.
```java
1 command = getClass("Command")
2 assert ( command.containsKey("execute()") | command.containsKey("wait()") )
```

In the next section, we describe DesignWizard — the API that we have developed to support design tests for Java language. Besides that, we describe how to use DesignWizard and JUnit framework to perform design verification.

III. WRITING AND EXECUTING JAVA DESIGN TESTS WITH DESIGNWIZARD

As we said before, to write a design test two components are necessary: a structure analyzer API and a testing framework. The former is responsible for: i) extracting information from code; and ii) exposing an API to retrieve this information. The last is responsible for: i) providing assertion routines to write design tests; ii) providing an automated way to execute the tests and iii) reporting test’s results.

To support design tests for Java language, we have developed DesignWizard, which plays the role of the structure analyzer API. Moreover, in our implementation of design tests for Java, we use DesignWizard combined with the most popular Java testing framework — JUnit.

For the sake of clarity, let us analyse Code 1 which is the Java implementation of the design test specified by the Pseudocode 1.

**Code 1** Design test implementation using DesignWizard and JUnit.
```java
1 public class DesignTest extends TestCase {
2   public void testPresentationRule() {
3       DesignWizard dw = new DesignWizard("project.jar");
4       Package pres = dw.getPackage("presentation");
5       Package data = dw.getPackage("data");
6       Set called = pres.getCalledPackages();
7       assertFalse(called.contains(data));
8   }
8 }
```

As we can see, DesignWizard’s constructor (line 3) receive a jar file representing the application under verification. At this time the API extracts the information about the structure of the application and the design is modeled as a set of entities and their relationships. This is a non-intrusive activity in the sense that the developer is not aware of how DesignWizard achieves extraction. Then, DesignWizard exposes methods to recover the information extracted. For instance, the method getPackage() (line 4) returns an object of type Package, which represents a specific package of the application under verification. Once the developer has a reference to this object, it is possible to invoke the method getCalledPackages() (line 6) in order to retrieve all packages that are called by a given one.

The last step of the test is the assertion phase. The assertFalse() assertion routine of JUnit framework is used to assure that the data package is not inside the set of the packages called by presentation.

Note that the design test is written entirely in Java language. Another important aspect to mention is that the design test is a JUnit test case (line 1). This enables the design test to be executed and reported in an automated manner reusing JUnit infrastructure.

DesignWizard provides a set of objects that gives information about the code under verification. The main abstractions are: Package, Class, Method and Field. The developer may use these objects to inspect a particular component of an application. For instance, it is possible to inspect a package using, among others, the following methods:

- Set(Class) getClasses();
- Set(Packages) getCalledPackages();

The method getClasses() may be used to acquire the classes that are defined inside a particular package. The other method, getCalledPackages(), returns a set of Package that represents the packages called by a particular one.
In the same way, it is possible to inspect structural properties of classes, methods and fields. For instance, Code 2 is the Java implementation of the design test specified by the Pseudocode 3. The code uses the method `extends()` (line 4) in order to check whether each class of package `command` is subclass of `Command` class.

**Code 2** Design test implementation using DesignWizard and JUnit.

```java
1 command = dw.getPackage("command");
2 Set classes = command.getClasses();
3 for ( Class c : classes ) {
4    assertTrue(c.extends(dw.getClass("Command"));
5 }
```

IV. Usability Evaluation

In a previous work [10], we have evaluated our approach regarding to its capability to detect divergences between implementation and design rules. In this study, among other aspects, we were interested in investigating the usability of DesignWizard API in order to claim that it has appeal to developers to perform structural conformance verification.

A. Methodology

To evaluate the API of DesignWizard, we have conducted an usability experiment using a methodology based on Microsoft usability studies [13] and successfully reproduced by other important researchers on this area [15], [16], [17].

According to Clarke [13], “It is the comparison between what developers expect and what the API provides that is interesting when evaluating the usability of an API”. So, it is clear that it’s important to capture the expectations of developers while using an API. We achieved this task using two strategies. The former was to ask developers to first write design tests pseudocode for a given design rule (before looking at DesignWizard API). In this case, pseudocode can give a realistic estimation on what methods developers expect to be found in DesignWizard. The later was to use the Think Aloud Protocol for APIs while developers were composing design tests using DesignWizard and the JUnit testing framework.

The Think Aloud Protocol for APIs is an approach to capture expectations of a developer about an API. The protocol encourages developers to verbalize their expectations, goals, difficulties and strategies when programming a specific task. This information is used to gather data about the behaviour of developers using the API. Then, this data is used to analyse whether the API meets their expectations. During the Think Aloud Protocol we have used screen capturing software to record the contents of the screen and programmers’ verbalizations.

Due to space limitation, we are not going to deeply explore DesignWizard. The full documentation is available on www.designwizard.org.

1) Environment: The experiment was performed in the Distributed Systems Laboratory from Federal University of Campina Grande (UFCG). For both, pseudocode and real design test writing phases, developers were presented with an Eclipse IDE environment and a browser with Javadocs. Each participant performed the experiment isolated from the others. At the beginning of the experiment, we gave a twenty minute talk to participants about design tests. To not compromise the validity of the experiment, it is important to point out that we did not mentioned at any time of the talk methods or abstractions of DesignWizard.

2) Project under verification: Participants were challenged to write design tests for real design rules in the context of the OurGrid project [18]. OurGrid is an open source peer-to-peer middleware for grid applications that has been in production since 2007. It is written entirely in Java language and contains 111,790 of LOC. Among other reasons, we have chosen OurGrid due to the fact that it had concrete design rules that must be followed by developers. Another reason is that OurGrid is located in the same lab that we work, which increased the contact with the team and made it easy to specify the rules to be verified. At last, it is important to point out that we are not part of the OurGrid team.

3) Participants: The experiment involved eleven developers with at least one year of experience in Java language. This constraint was imposed to avoid beginning Java developers. The eleven participants had between two and seven years of Java experience, with a median of four years. Among them, five developers are undergraduate students, while the remainder are graduate students. All participants were male, and ranged in age from 20 to 27, with a median age of 22. It is important to note that none of the participants had previous experience with design tests or DesignWizard.

4) Design Tests: We have challenged eleven developers to write five design tests that regard the OurGrid design rules. The rules were selected after a meeting with the leader of the OurGrid who specified what were the major design constraints of the project. The design rules were selected to meet the two following criterias:

- Explore different types of design rules;
- Explore several abstractions of DesignWizard API.

The first criterion regards the fact that it’s important to explore design tests with respect to its capability of express several types of design rules. The second criteria was chosen to make developers manipulate several abstractions of DesignWizard to write the design tests. The following design rules were proposed to be written as design tests:

**Rule 1 - DAO-Controller**

*Only org.ourgrid.peer.controller package may access the org.ourgrid.peer.dao package.*

This is the simplest design rule. It regards the communication between two specific packages from the OurGrid...
code. In spite of its simplicity, this rule plays an important role in the context of the OurGrid, since it intends to restrict the access to data objects.

**Rule 2 - Cyclic Dependency**

*There must be no cyclic dependency between packages.*

Cyclic dependency between packages happens when a package A depends on package B and package B also depends on package A. This rule also regards to communication between two packages of the code. However, unlike Rule 1, this rule does not rely on specific packages. The developer must be able to inspect all packages and check them. The importance of this rule relies on the fact that cyclic dependency makes code harder to understand, modify and test.

**Rule 3 - Common Package**

*All classes from org.ourgrid.common package must be used at least by two different packages.*

This rule explores not only the abstraction Package from the API, but also the abstraction Class. Besides that, the developer needs to inspect the classes inside a package. According to the leader of the OurGrid project, this rule is important to keep in org.ourgrid.common package only those classes that are really of general interest.

**Rule 4 - SchedulerCommandExecutor Class**

*All classes from org.ourgrid.executors package must extend SchedulerCommandExecutor class.*

This rule relies on a specific relationship between classes of the code. That is, the developer must be able to inspect the inheritance relations between classes. The leader of the OurGrid project reported that command classes (classes that extends SchedulerCommandExecutor) play an important role in the project. For this reason they must be inside a specific package.

**Rule 5 - HashCode and Equals**

*All classes from the code must override the hashCode(int) method and equals(java.lang.Object) method.*

Rule 5 is important to be verified due to the fact that some data structures of Java language use both methods to verify the presence of an object. Hence, a common good defensive programming practice is to override the hashCode(int) and equals(java.lang.Object) methods for all classes.

B. Study Results

Through the data collected from the Think Aloud Protocol and the comparison between developer’s pseudocode and DesignWizard API, we can point out three main observations:

1) The DesignWizard meets the expectations of developers;
2) Developers had no difficulty to write design tests;
3) In our approach, the language used to specify design rules is not an obstacle.

These observations are supported by the results of all design tests. Let us analyse them separately:

**Rule 1 Results: DAO-Controller.** Table II details the results from the comparison between the pseudocode produced by developers and the DesignWizard for the Rule 1. This comparison quantifies the expectations of developers and describes whether the API meets or not these expectations.

<table>
<thead>
<tr>
<th>Expected method</th>
<th>#PO</th>
<th>Present in DW?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPackage(String name)</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>getCallers()</td>
<td>8</td>
<td>Yes</td>
</tr>
<tr>
<td>getAllClasses()</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>getAllPackages()</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>getName()</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>new PackageNode(String name)</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>getAccessingPackages(package)</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

Among other steps, to write a design test for Rule 1, it is necessary to get information about a specific package of the code. To achieve this, developers adopted three different strategies. The first one, adopted by nine of the eleven developers, consisted in using the method getPackage(String name). The second one, adopted by one of the developers, involved iteration over all packages (getAllPackages()) and comparison through the name of the required package using the getName() method. At last, one of the developers tried to create an instance of Package abstraction. Among the three strategies adopted by developers, only the last one is not possible to achieve using our API. That is, only one developer specified on pseudocode a method that not appears in DesignWizard. Nevertheless, the Think Aloud Protocol revealed that all developers specified correctly and without difficulties the design test using DesignWizard and JUnit.

**Rule 2 Results: Cyclic Dependency.** Table III details the results for Rule 2. Again, it is possible to note that DesignWizard meets the expectations of developers in most of the cases. However, the data analysis revealed that four developers were expecting a method calls(package) in the DesignWizard. Although these developers were not satisfied by the API, the Think Aloud Protocol revealed that they easily found the alternative provided by the DesignWizard, which includes calling the method getCalledPackages() and verify whether the target package is inside the returned collection.

All participants correctly specified the design test using DesignWizard and JUnit for Rule 2. An important observation, extracted from the Think Aloud Protocol, is the fluency...
of developers when they are composing a design test. This happens because they do not differentiate the act of writing functional code from the act of writing design tests. In fact, both activities involve only Java language. Thus, the language used is not an obstacle to specify design rules.

### Rule 2 Results. \#PO = Pseudocode Occurrences.

<table>
<thead>
<tr>
<th>Expected method</th>
<th>#PO</th>
<th>Present in DW?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCalledMethods()</td>
<td>10</td>
<td>Yes</td>
</tr>
<tr>
<td>getCallers()</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>getClasses(PackageNode)</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>getClasses(String regex)</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>depends(package)</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

### Rule 3 Results: Common Package

For this rule, only two developers expected methods that were not in DesignWizard. However, these two developers found the alternative provided by the API. The results of this rule follows the tendency from the other rules, which indicates that, even if the API does not provide all methods that developers expect, they can easily find alternatives to write the design tests.

### Rule 4 Results: SchedulerCommandExecutor Class.

The analysis of the design tests composed to specify this rule revealed an important absence in DesignWizard. In order to specify Rule 4 as a design test, it is necessary to check whether a class inherits from another one. As we can see in Table IV, eight developers expected the method `extends(Class class)` to be in the API in order achieve this task. However, the API of DesignWizard does not provide this method. Despite to this fact, all of these developers found the alternative of using the `getSuperClass()` method, which reinforce our observation that DesignWizard has an attractive usability, once we observed scenarios that, even in the cases that the API did not meet the developer’s expectations, participants easily found an alternative provided by the API.

Unlike the results of other rules, there were errors in the design tests composed for Rule 4. Six developers expected that the method `getAllClasses()` returned a set containing only objects that represents classes. However, this method returns not only classes from the code being extracted, but also the interfaces. We have adopted this nomenclature to be in conformance with Java Reflection API, which is the standard Java API to dynamically retrieve information about the code. It became clear to us that the name of the method confused the participants. Nevertheless, a simple inspection in the javadoc of the method could have solved the problem. During the Think Aloud Protocol, it was possible to identify that only one developer read the documentation and, as a result, correctly wrote the design test.

### Rule 5 Results: HashCode and Equals.

The results of this rule also consolidate that our API meets the developers expectation in the sense that it contains the expected methods to compose a design test. For this rule, only one developer expected a method that was not present in the API, which was not an obstacle to write the design test using DesignWizard.

In the same way that Rule 4, although all participants could write the design test, five developers did not write correctly. Again, participants used the correct method, but they did not correctly used it. In order to acquire information about a declared method of a class, DesignWizard provides the method `getDeclaredMethod(String name)`. The misunderstood was caused by the nomenclature used by us in our API. The parameter to be passed by the method is called `name`, but an appropriated parameter name would be `signature`, once this information is what uniquely identify a method in Java.

### C. Discussion

From a research view, it is important to say that we do not intend to generalize the observations found. It is obvious that it is not possible to guarantee that most of Java developers will easily manipulate the API to write design tests. Nevertheless, the results from our study are strong indications that our API is easy to use. We systematically evaluated our API with real world design tests and with its target clients — developers. We believe developers behaviour and opinion are two of the most important source of information about the usability of an API. Thus, we believe our results and observations are reliable from a research perspective.

One important number that advocates in favor of the DesignWizard usability is that all developers were able to implement all the proposed design tests. Nevertheless, eleven design tests were not appropriately composed. In all cases, the proper method was correctly chosen but was not used in a properly manner. The results leaded us to change a method name and improve de javadoc quality.

Another important observation that we collected from the experiment is that developers prefer assertive boolean methods rather than query methods to assure a specific property. For instance, in place of using the method `getSuperClass()` and inspect for a particular class, developer do prefer to call `extends()` method. As a consequence of this observation, we have added some assertive boolean methods to the API.
The Think Aloud Protocol is interesting to investigate the developers’ behaviour while using an API. We observed that developers spontaneously used the features of Java language to achieve design test compositions. This scenario is supported by the data collected from the Think Aloud Protocol and the developers’ code. For example, to inspect whether a package contains a given class, developers adopted the strategy of retrieving all methods of the given package and using the method `contains()` of Java Collections Framework. The same observation is supported by the use of methods like `equals()` and the structure of the language such as `for` and `if` commands, and logical operators.

In particular, we have observed that designers and programmers appreciate the use of design tests as an executable documentation that can be easily kept up to date. Tests are usually easy to write, to read and to maintain. Furthermore, one can easily decide the level of rigor necessary to apply to each part of the software, promoting the appropriate balance of rigor and freedom.

V. ESCALABILITY EVALUATION

An important requirement of our solution is that it must scale as the size of software under verification grows. One of the problems of manual code inspections is that, as software grows, it becomes harder to manually inspect for design rules violations. For this reason, we have performed an experiment in order to show that time and memory spent by DesignWizard to perform verification are not prejudicial to the adoption of our approach.

Scalability evaluation was carried by gradually increasing the size (from 0.1 to 46MB) of eight jar file project under verification, shown in Table V. It is important to note that the jar files of the projects were pre-processed to contain only `.class` files, which are those extracted by DesignWizard. External Tools Plugin and Java Standard Library were chosen to represent small and large projects respectively. FindBugs (2.7MB) was chosen due to its representative size, since it is equal to the medium of the sizes of the ten most popular Sourceforge projects.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size (MB) of Jar file</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Tools Plugin</td>
<td>0.1</td>
</tr>
<tr>
<td>Subversion Plugin</td>
<td>0.2</td>
</tr>
<tr>
<td>Text Editor Plugin</td>
<td>0.5</td>
</tr>
<tr>
<td>Team UI Plugin</td>
<td>1.1</td>
</tr>
<tr>
<td>FindBugs</td>
<td>2.7</td>
</tr>
<tr>
<td>JDT Plugin</td>
<td>8.2</td>
</tr>
<tr>
<td>Azureus</td>
<td>11.7</td>
</tr>
<tr>
<td>Java Standard Library</td>
<td>46</td>
</tr>
</tbody>
</table>

Figure 3 presents the evaluation results in logarithmic scale for the CPU time consumed by DesignWizard to apply static analysis on the projects. As we can see, CPU time is linear function of the size of software under verification.

To demonstrate that, we have performed a linear regression analysis and the coefficient of determination points to a strong linear association ($R^2 = 0.9946$).

![Performance Evaluation](image)

Analysing Figure 3, it is safe to say that efficiency is not a problem for DesignWizard. Even the time of Java Standard Library extraction is acceptable, since it can be considered a large project. Note that for FindBugs, which has a representative size of an application, the CPU time is only 2.7 seconds.

Figure 4 presents the evaluation results for memory consumption. Once again, the variable studied is linear function ($R^2 = 0.9892$) of the size of software under verification. We have used JProfiler [19] to measure the memory usage.

![Memory Usage Evaluation](image)

It is important to point out that we had to increased the available memory by the Java Virtual Machine in order to complete the extraction for Java Standard Library. However, this is a common practice to run large Java applications. If
this is considered to be a bottleneck, one possible solution is to proceed the extraction by steps and save the data on a database instead of maintain in memory.

One possible bottleneck is the re-analysis that each execution of design test causes. However, this issue can be easily addressed by saving the result of an extraction and performing re-analysis only on those classes that have changed between two versions of a jar file.

The experiment was executed in a Intel(R) Core(TM)2 Duo 2.33GHz with 1GB of memory, running Linux Kernel 2.6.28-11-vserver Debian distribution.

VI. RELATED WORK

Conformance checking between code and design is one of the most important activities to guarantee quality on software. Typically, design specification is achieved as a result of important decisions regarding quality aspects about the structure of the code. Hence, decisions made at the design specification phase are crucial for software quality, maintenance and evolution. For this reason, it is important to keep the code being developed synchronized with the specified design.

In a brief statement, design conformance checking is a comparison between an implementation and a design documentation reference. Often, design specification is a set of design rules that relies on structural relations between the components of the software. There have been several attempts to apply structural conformance checking. For example, Design Review [6], which is a particular type of code inspection, consists in manually examine the code in order to find a set of design issues that includes divergences between the specified design and the implemented code. The problem with Design Review is that, being a manual process, that can lead the team to commit errors during analysis. Moreover, as software evolves, it becomes harder to check conformance, since analysing several classes may take several hours [20], [21].

That are also approaches proposed to check code against design rules in an automated manner. ArchJava [4], for instance, is an extension of Java programming language that aims to ensure that the implementation conforms to architectural decisions. This extension is used to specify architectural constraints by using the concepts of ports and components introduced by the authors. This work is strongly related to ours in the sense that it aims to check the conformance between the code and specified rules. However, besides high level design rules such as integrity communication, our approach relies in many other low level design aspects of the code, like inheritance between components, field access and method calls. Besides that, ArchJava requires a compiler and learning the syntax of the extended Java language to specify the constraints, while our approach does not require effort to learn a new language to write the design rules.

Murphy et al. [3] developed a software reflexion model technique to help engineers to perform various software engineering tasks by exploiting the drift between design and implementation. This is an interactive approach. Developers need to specify the expected model and describe a mapping between the extracted source model and the stated high-level structural model. This work uses a sequence of reflexion models to compare architectural design of a program with its source code.

There are also tools which aim to automatically check structural properties of code. FindBugs [22], for example, is a tool that uses static analysis to discover bugs on Java code in an automated manner. The tool analyzes the bytecode of a Java application and generates a report containing the probable bugs found on the analysis. LClint [23] works similarly, however, it is 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. By the other hand, we focus on design level.

VII. CONCLUSION

This paper has dealt with structural conformance checking using design tests — an approach whose main goal is to automatically check whether implementation is in conformance with design rules. Design tests are automated tests that specify design rules relying on structural properties of the code. Besides the concept of design test, we also have described in this paper DesignWizard, an API that gives support to write design tests for Java applications. DesignWizard extracts information about the code under verification and exposes methods to recover this information in order to support developers to write design tests. The tests are specified using DesignWizard and JUnit, which enables an automated way to perform and report the verification.

We have evaluated DesignWizard regarding to its usability. We have challenged eleven developers to write five real world design tests for OurGrid — a Java grid middleware with 111,790 LOC. We have used Think Aloud Protocol for APIs in order to catch developers experience with our API. The results lead us to conclude that developers had no adversities while composing design tests with DesignWizard. Moreover, we have observed that the API meets the expectations of developers in the sense that it provides the expected methods to write a number of design tests.

We also have evaluated the scalability of DesignWizard. We gradually increased the size (from 0.1 to 46MB) of eight jar file project under verification in order to investigate CPU time and memory consumption behaviour. The results shows that both are linear functions of the size of software under verification.

We are currently constructing a design test catalog to check several good general programming practices. We intend to share this catalog to encourage the adoption of our
approach and that can be used as a reference for those who wants to compose design tests. As a future work, we would like to extend DesignWizard in order to provide support to write design tests that relies on dynamic properties of the software.

VIII. ACKNOWLEDGEMENTS

We would like to thank Rodrigo Vilar for his patience in explaining the design rules that were used in our evaluation. Special thanks go to all participants of the usability experiment. This work has been partially supported and developed in collaboration with CPM Braxis and FINEP/MCT/Brazil through grant number 3156/06.

REFERENCES


