Design Tests: An Approach to Programatically Check your Code Against Design Rules

João Brunet, Dalton Guerrero, Jorge Figueiredo
Federal University of Campina Grande
Department of Computer Science
Campina Grande - PB, Brazil
{jarthur,dalton,abrantes}@dsc.ufcg.edu.br

Abstract

Assuring that a program conforms to its specification is a key concern in software quality assurance. Although there is substantial tool support to check whether an implementation complies to its functional requirements, checking whether it conforms to its design remains as an almost completely manual activity. In this paper, we present the concept of design tests, which are test-like programs that automatically check whether an implementation conforms to a specific design rule. Design rules are implemented directly in the target programming language in the form of tests. As a proof of concept, we present DesignWizard, an API developed to support design tests for Java programs as JUnit test cases. We applied design tests in two case studies and observed that our approach is suitable to check conformance automatically. Moreover, we observed that designers and programmers appreciate design tests as an executable documentation that can be easily kept up to date.

1. Overview and Motivation

Checking conformance between implementation and design rules is an important activity to guarantee quality on source code. One process widely used by software companies is Design Review [9], which consists of a mechanism to find errors in the design and its representation by splitting the team into groups and making them analyze each others’ code. The problem with Design Review, being a manual process, that can lead the team to errors during the analysis. Moreover, this process does not scale, since analyzing a number of classes may take several hours.

There have been several attempts to check conformance between code and design rules automatically. ArchJava [1], for example, proposes an extension of Java programming language that aims at ensuring conformance between implementation and architectural constraints specified using the concepts of ports and components introduced by the authors.

Fairbanks et al. [4] use the term design fragment to refer to patterns that describe how a program interacts with frameworks. Design fragments describe what programmers should build to accomplish some goals and which are the relevant parts of the framework that the programmer’s code will interact. The authors have created a catalog of design fragments and also mechanisms to implement (using XML) design fragments to check conformance between application and those design fragments.

Another related work [8] describes a software reflexion model technique to help engineers to perform various software engineering tasks by exploiting the drift between design and implementation. This approach is interactive and the user has to specify the expected model, extract the real model from the source code using static analysis and then, describe a mapping between the extracted source model and the stated high-level structural model.

Despite the existence of several academic approaches to check conformance between code and design rules, the gap between the state of the art and the state of the practice has become most apparent. We believe one of the factors responsible for this scenario is that these approaches generally require learning a language different from that of software under analysis to describe the design rules. This increases the learning curve and put at risk the adoption of these approaches. Design Fragments proposed by Fairbanks et al. are an important step towards an increase in the adoption of conformance checking, since the authors have built an infrastructure to apply their approach on the developer’s environment. However, we believe that an approach that allows specification of design rules in the same language as that of the software under analysis and as automated tests can increase the adoption of conformance checking by both designers and programmers.
In this paper, we present the concept of **design tests**. A design test is a test that checks whether an implementation complies with a given design rule expressed as an algorithm. Design tests differ from functional tests because the latter ones are used to check whether the software does what it is supposed to do. On the other hand, design tests check how the software has been built.

Design tests are automated tests. They can be written using existing testing harnesses in the **same programming language used to develop the software under test** (we use JUnit [6] in our Java implementation, discussed later). This has the benefit that design rules can be documented in a language that all developers understand. Furthermore, as an executable documentation, design tests are more likely to keep in synchronization with the actual implementation. We believe this is especially useful in development teams in which developers change often and can easily miss or misunderstand design decisions.

To support the use of design tests, we have developed DesignWizard (DW), a fully fledged API that allows writing design tests for Java implementations, using the JUnit testing framework. We have applied design tests in the development of at least three software products we have developed in our labs at UPCG. We have also used design tests to assess student projects in undergraduate courses. Our experience has shown that using design tests is easy and can be easily incorporated into a development process to improve software quality. 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. Finally, we have observed that the implementation is scalable and that efficiency has not been a problem.

The rest of the paper is structured as follows. In Section 2, we describe our approach and how it must be applied to check the conformance between implementation and specified design rules. Section 3 describes our experiences in applying design tests in two case studies. Finally, Section 4 concludes the paper with our final remarks and a brief discussion on future work.

### 2. Design Test

The key concern of our approach is the construction of a **design test**, in which design rules are specified as algorithms and then checked in an automated way. To accomplish this, a designer uses two key components: a **code structure analyzer API** and a **testing framework**.

The code structure analyzer API is responsible for: i) Extracting facts from code; and ii) Exposing an API to re-cover this information. This component applies static analysis to extract information about the structure of the code under analysis. The facts extracted must to be modeled as a graph, where the nodes are the entities and the edges are relations between them. Finally, this component provides an API that offers methods to navigate through this graph.

The testing framework is responsible for: i) providing assertion routines; ii) providing an automated way to execute the tests and iii) reporting test’s results. In our approach, the assertions routines will be used to check not the functionality of the code, but its structure.

Pseudocode 1 illustrates an example of design test performed in OurGrid [2] (one of our case studies). It checks whether the classes inside the **dao** package of the **peer** component are only called by the classes inside the **controller** package of the same component. In other words, any attribute access or method call regarding the **dao** package classes must come from the **controller** package or from the **dao** package itself. Checking this rule avoids unnecessary direct coupling between presentation objects and data objects, making the code easier to modify and maintain.

#### Pseudocode 1 Design test pseudocode.

```java
1 daoPackage = org.ourgrid.peer.dao
2 controllerPackage = org.ourgrid.peer.controller
3 FOR each class in daoPackage DO
4    callers = class.getCallers()
5    FOR each caller in callers DO
6       assert caller in controllerPackage || caller in daoPackage
7   ENDFOR
8 ENDFOR
```

The result of the execution has a different interpretation from the results a functional test. A success in the execution of a design test means that the code analyzed follows the design rules described. By the other side, a failure denotes that the code does not follow the specified design rule.

In our implementation of the design test approach, DW plays the role of the code structure analyzer API, while JUnit has been used to provide assertions, execute and report the results of a design test execution.

### 3. Evaluation

In this section we present our initial experience in applying the design test approach in OurGrid and OurBackup [3] projects. Additionally, we present DW’s performance evaluation results and some limitations of our approach. Our focus in this initial evaluation was to quantify violations of

---

1DesignWizard is an open source software and is distributed under the GNU LGPL license. It can be freely downloaded at http://www.designwizard.org
design rules, to investigate how to deal with violations and to investigate whether design tests are easy to compose and understand. To accomplish this, we conducted experiments on two different scenarios. In the first one we composed a design test to be applied in OurGrid, executed the design test, quantified the violations, presented the violations to the leader of the project and presented the design test to the development team. The second scenario differs from the first because the leader of the OurBackup project composed the design test, not us. OurBackup is a solution for backup on P2P systems containing 133,974 LOC, while OurGrid is a free to join P2P grid with 111,790 LOC. Both projects were developed at the Distributed Systems Labs at UFCG.

In OurGrid case, we have applied our approach by executing the design test described by Code 1, which is the implementation of the Pseudocode 1 using DW and JUnit. The design test execution revealed 10 violations which were reported to the leader of OurGrid development team. It was presented to four developers of OurGrid. All developers understood what the test was supposed to do and appreciated the way to check whether they are following the rules.

Code 1 Design test Java code for dao package access.

```java
public class OurGridDesignTest extends TestCase {
    public void testCommunication() {
        DesignWizard dw;
        dw = new DesignWizard("ourgrid.jar");
        PackageNode dao = dw.getPackage("org.ourgrid.peer.dao");
        PackageNode controller = dw.getPackage("org.ourgrid.peer.controller");
        Set<ClassNode> callers = null;
        for (ClassNode clazz : dao.getAllClasses()) {
            callers = clazz.getCallers();
            for (ClassNode caller : callers) {
                assertTrue(caller.getPackage().equals(dao) || caller.getPackage().equals(controller));
            }
        }
    }
}
```

On the other scenario, the leader of OurBackup development team constructed a design test to check whether the test facade has the same methods of the system facade. According to him, this is an essential rule because the qualifying team of OurBackup is different from the development team. The qualifying team only knows the test facade of the system and they have to test OurBackup through this facade.

In this case, no violation was identified. The leader of OurBackup reported that composing a design test was not hard to do because he only had to learn an API, which is a task that is common for programmers. He also reported that design tests are an useful technique to assure that his decisions are being followed without having to manually analyze it and design rules expressed as an algorithm facilitates the communication of low level design decision between him and the development team. The design test was presented to others members of the development team and they highlighted that one of the main benefits of our approach is that they can read and check the rules in their environment, avoiding the use of external tools or different specification languages to check conformance.

**Discussion.** According to the leader of OurGrid development team, three of the ten violations identified were acceptable. For example, the classes inside test packages need to access dao package to verify the functional correctness of the code. In this case, the rule was modified to include the org.ourgrid.test in the list of packages that are allowed to access the dao package. The other two acceptable violations were related to the common package that, in the context of OurGrid, is allowed to access dao package. However, the other seven violations identified were considered critical for the software evolution. For example, the class WorkersTable, which is part of the GUI code, access the class WorkerEntry inside the dao package. According to the leader of OurGrid development team, this violation is unacceptable, since GUI code should never access dao objects directly.

The use of the design test approach is specially important in projects that use tests to assure quality of the software. The development team of OurGrid cannot commit code without executing all functional tests (both unit and acceptance) and getting a successful result. If the design test we have presented was added to their test suite, they would have an automated manner to check whether the code follows the specified design rule. This way, If the code written does not pass in the design tests, the programmer would not be able to commit it, even if the code follows its functional requirements. This can reduce the coupling between unrelated packages before code decay, minimizing the effects of design erosion [10].

Although no violation was identified on OurBackup, the conformance checking between the facade design rule and the implementation was performed manually before design tests. However, in the current implementation of OurBackup, the design test was added in the test suite in order to be performed daily.

**DW Performance Evaluation.** To show that DW scales well, we conducted an experiment to assess its performance. We measured the time that DW takes to complete the static analysis on the bytecode of four projects with size between 224KB (DW) and 46MB (Java Standard Library), which were chosen to represent small and large projects respectively. FindBugs (3MB) was chosen due to its representative size, once it is equals to the medium of the sizes of the ten most popular Sourceforge projects in 10/01/2008.

Figure 1 presents the evaluation results. As can be seen, efficiency has not been a problem, since the time of extraction for FindBugs is acceptable. Even the time of Java Stan-
standard Library extraction is acceptable, given that it can be considered a large project. One possible bottleneck is the re-analysis that every call of DW’s constructor 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. It is important to note that all jars analyzed only contain class files, which are those processed by DW. The experiment was executed in a Intel(R) Core(TM)2 Duo 2.33GHz with 2GB of memory, running Linux Kernel 2.6.24 Debian distribution.

One limitation of our approach is that it’s not possible to specify design rules that rely in dynamic aspects of the code. For instance, it is impossible to create a design test to check whether the method stop of an object is always called after the method start. Additionally, our current implementation does not cover design rules that rely in threading behavior.

4. Conclusions and Future Works

We have presented an approach whose main goal is to enable an automated manner to check whether implementation is in conformance with design rules. This is achieved by composing design tests - a kind of test that relies in design aspects of the code. To apply our approach, we have introduced DesignWizard. We have also demonstrated with empirical examples that DesignWizard can be used in real systems such as OurGrid and OurBackup. DesignWizard has been used for several purposes, including automated code generation and impact analysis [5]. It is currently in version 1.4 and, as far as we know, nine projects are currently using it. These projects are in the context of academic researches and software companies such as CPM Braxis in Brazil.

Our main future work is to apply a complete evaluation of the design tests approach. The results spread through this document are indications that our approach can be a very attractive alternative to check conformance between design rules and implementation.

Another future work is to construct design tests for rules related to design patterns and frameworks. We also plan to compose a design test catalog to check several good programming practices, such as avoiding method invocation on constructor and assuring that methods follow the law of demeter [7]. This catalog will be shared with the community in order to be used without any effort, but executing the design tests. We believe that this catalog will encourage the adoption of design tests.

5. Acknowledgements

This work has been partially supported and developed in collaboration with CPM Braxis and FINEP/MCT/Brazil through grant number 3156/06. Special thanks goes to Ayla Dantas for providing ideas and promoting relevant discussions about this work.

References