

# Improving Code Quality on Automated Tests of Web Applications: A Set of Patterns

Maurício Finavaro Aniche, University of São Paulo / Caelum Ensino e Inovação, Brazil  
Eduardo Guerra, National Institute of Space Research (INPE)  
Marco Aurélio Gerosa, University of São Paulo, Brazil

**Abstract.** *There are different levels of automated testing, such as unit, integration or system testing. When dealing with web applications, the act of writing automated tests usually requires a huge effort from developers, like tests that usually open the browser and navigate through the web page. However, the more integrated the test, the more complicated it is to write an easy to read and maintain test code. In this paper, we present a set of patterns that deals with real world problems when writing automated tests for web applications, such as the difficulty in building scenarios and the constant change of the HTML element's attributes.*

## 1. Context

The use of automated software testing has been growing in popularity for the last decade. And, in fact, it does make sense. Many studies have shown that automated testing reduces the number of bugs and, thus, it improves external quality [Janzen, 2005] [Maximilien, 2003] [Nagappan, 2006].

As a consequence, the quantity of test code in software systems has grown substantially. So, a new requirement for developers is to write maintainable test code. However, this is not an easy task. Each level of automated testing contains its own problems and difficulties. Unit testing, as an example, requires the use of Mock Objects [Freeman et al, 2004], which makes the unit test code more complicated.

Testing web applications can be tricky. Besides unit testing, developers also write system tests. System tests are black-box tests, in which the application is tested as it is, with all components working together. In a web application, it implies opening the web browser and navigating through the web page.

Many problems appear when dealing with real web applications. To test AJAX requests, for example, the system test should wait while the request is being executed. In addition, there must be a timeout, as the test does not know if the request is still being processed or if the system has crashed. Also, developers keep changing HTML elements and their properties. To keep tracking all of the scenario that a test needs is also complicated.

In this paper we deal with some of these problems. We present a few patterns in which the goal is to help developers to write a more maintainable test code. They were extracted from a real

case study. Alura [Caelum, 2012] is an e-learning platform, built by Caelum<sup>1</sup>. The team has been developing this software for the last two years. It is a Java web application, which contains around 750 classes, 800 unit tests and 30 web tests. The team contains from 3 to 5 members, depending on the importance of the project at the moment. Their experience in software development varies from 2 to 10 years.

## 2. Web Service API for Building Scenarios

*In order to test a specific feature, the web application must be in a desired state prior to the execution of the test. However, building the specific scenario can be hard as the scenario may be complex and depend upon many different entities.*

\* \* \*

**What is the best way for a developer to populate web applications with the specific scenario that is required by a system test?**

Dealing with scenarios in system tests (not only on web applications) is different from what happens in unit tests. When writing isolated tests, scenarios are basically a set of instantiated objects with their properties set to what the test requires. However, a web application test usually consumes an application that is running on a real server with real databases and/or web services and so on. So, building up a scenario means to insert data on whatever the web application retrieves it from.

There are a few ways to do it. One of them is to make automated test to navigate on the interface and create the data through the interface. Suppose a CRM application, in which, to test the "Order" feature, a Customer needs to exist. The test can navigate to the "Create Customer" feature and make use of it. And then, after the customer was added, the test can now test the "Order" feature.

The problem with this approach is that (i) depending on the scenario, the test will need to navigate through many different pages, (ii) if the "Customer" feature breaks, the "Order" can not be tested and (iii) the total time of the test execution will increase.

Therefore:

**Treat an API that is accessible through any common protocol, such as REST or SOAP, that builds up any scenario that is required by the automated test.**

Developing web services is not a complicated task and there are many different libraries in every programming language that facilitates the job. In order to build an useful API, developers should

---

<sup>1</sup> <http://www.caelum.com.br>. Last access on May, the 21th, 2014.

work on the server and on the client side of the API.

The server side API must be able to receive requests with a single (or a set of) entity (entities) and persist with it into the system. The content of the request must be in a common format, such as XML or JSON. Depending on the web framework the web application uses, this is a common behavior: the action supports many different formats, depending on the CONTENT-TYPE that was sent on the header of the HTTP request.

So, for an imaginary "Customer" entity, the server side must support the inclusion, deletion and update operations. If any other operation on the entity is required, it must be developed by the team. We argue that it is not a problem to create web services only for testing purposes.

On the other hand, the client API must facilitate the access to the server API. It means that it should be easy to pass entities and let the API deal with problems, such as the serialization of the data and the HTTP request. In Listing 1, we show an example of a real API that is used to create a user that has a subscription on the system. The class `UserAPI` is responsible for consuming the web service that deals with users.

---

```
@Test
public void should_answer_an_exercise(){
    User john = new User("John");
    new UserAPI().addUserWithSubscription(john);
    DashboardPage dash = login.logAs(john);

    // the test continues here...
}
```

---

Listing 1. An example of the client API that consumes the server API.

If these web services are only for testing purposes, your software should hide the API when in production and allow the API on testing environment. Developers should not be afraid to write testing code or APIs that facilitate the test; in fact, the more flexible the API is, the easier the developers will write system tests. So, working on a base code/internal framework that helps doing it may be a good idea.

\* \* \*

Another possibility would be to make the test to connect directly to the server database and make use of all *Test Data Builders* and *DAOs* to create the scenario. In this case, there is no need for web services<sup>2</sup>.

---

<sup>2</sup> This can become a new pattern in the future.

Also, many developers make use of DBUnit<sup>3</sup>, a JUnit extension that puts the database on the desired state before the execution of the test. The framework is based on XML files, which are loaded to the database. When dealing with stable systems (and by that, we mean systems that do not change frequently anymore), loading XML files may be a simple solution to the problem. On the other hand, if the system and the database schema change frequently, it is harder to maintain the XML file than the suggested web services. Do not forget to take the stability of the system into account, when deciding how to build the scenario.

*On Alura (Caelum, 2012), after many different attempts of building up scenarios to the web application, the team decided to build up the API. The API makes use of REST and XML to exchange data. Some of the web services are also used in production; some of them only exist for testing purposes. Nowadays, through the API, developers can create any scenario required by the test.*

### 3. Database Always Clean

*Automated tests are sensitive to the information that exists on the system. When building up scenarios in system tests, it is common that the scenario keeps persisted on the system, even after the execution of the test. Having a dirty database can make your test fail.*

\* \* \*

#### **What should developers do to reduce the influence of one test scenario on another?**

A test should not depend and/or influence another one. When this rule is not followed, a test may fail, not because the tested behavior is not working anymore, but because the other test left the application in an invalid state.

Resetting test scenarios in unit testing is simple: all objects get destroyed as soon as the test finishes. However, in web testing, the scenario that is created by one test usually persists even after the test finishes. It does make sense, after all, the web application uses a real persistence mechanism.

As said, having unexpected data on the application will probably cause a test to fail. Suppose a test which guarantees that a report about the total sales amount works. If there are more sales than what the test expects, then the test will fail.

In addition, writing a test that would understand the current scenario and recalculate the expected result (or even change the existent state to the expected one) is hard to be written.

---

<sup>3</sup> <http://dbunit.sourceforge.net/>. Last access on June, the 20th, 2014.

Therefore:

**Clean all the data that is stored on any persistence mechanism that the web application uses before the test runs.**

If the tested web application is always clean before the execution of the test, then the test will not fail because of any dirty data. However, cleaning up data in web application is a problem as complicated as to insert data on it.

The web application must provide a "backdoor" that erases all data. It must be enabled only in the testing environment and should be responsible for putting the system to the simplest state that it can be. The simplest state, sometimes, may not imply deleting the content of every table on the database. Imagine a system that deals with delivering products in a country. The table with all the cities must be already loaded in the system prior to any test. So, the service should populate the list of cities as well as any other required information.

Depending on the size of the test suite, it is common for the team to parallelize the execution. However, as the database is truncated before every test, a drawback of this approach is that it is not possible to run many tests concurrently against the same server. A possible solution would be to deploy more than one test server and balance the execution of the tests among the machines.

\* \* \*

The Web Service API for Building Scenarios can also provide an API to reset the web application to the initial state.

*On Alura (Caelum, 2012), before any test starts to run, the reset API is invoked, and it puts the system to its initial state. In that case, all data is deleted, with the exception of administrator users. The system requires at least one administrator to work, so it is created by the API.*

## **4. ID to All Testable Elements in HTML**

*Web testing deals with HTML elements all the time. However, as a natural instability of the user interface, it is common that a test stops working because of a change on some HTML element's property. A fragile test can increase the maintenance cost of the test suite.*

\* \* \*

**How should a developer search for an HTML element in an automated test, in a way that s/he does not have to change it anymore?**

Any web testing API allows developers to search for elements on the HTML page. To do that, developers should tell the API how the element should be found. The element can be found by looking at its CSS classes, at its ID, at its name, or even at its relative position in relation to other elements.

However, the user interface (UI) tends to change a lot during the website's life cycle. That is why an automated web testing tends to be highly unstable. To implement these changes, developers commonly change CSS classes and even the position of the element on the page. As soon as something changes, let's say, a CSS class, if that test used that class to search for the element, the test stops working.

Although the CSS class or the position of an element can be changed, a developer rarely changes its ID. After all, most of Javascript binding and events are based on the element's ID. Also, the ID of an element is unique on that page, which guarantees that the right element has been found.

Therefore:

**Locate all elements in the HTML through the element's unique ID, and not through their CSS class, relative position, or any other way of finding an element in the page. If the element does not have an ID, put an ID just for the test.**

If, for some reason, an element has a very unstable ID (it may be dynamically generated), developers should create any specific property for the testing. HTML5 allows developers to create any extra attribute on any HTML tag. In Listing 2, we show an example of a specific attribute, called *data-selenium*, which is used only for the test.

---

```
<input type="text"
      id="customer_${i}"
      name="customer"
      data-selenium="name" />
```

---

Listing 2. An HTML element with an extra attribute to allow testability.

If one thinks that this extra property may be a problem in the production environment, it can be removed during the production deployment package generation. There are many tools that manipulate the HTML pages before deploying it (minification is an example of that).

*On Alura (Caelum, 2012), there were many tests that used to stop working because of some change on some element's property. Then, the team decided to search elements through its ID or, in the worst case, to create a new attribute. Tests became much more stable after that.*

## 5. Page Objects All the Time

*A software is usually compound by many features (functionalities). Features are highly connected on web applications. It means that a test for a specific feature may have to deal with the page that belongs to another feature. Repeating code is always a bad idea.*

\* \*

**How can we reduce the number of points to change when the feature changes and reuse that code over many different tests?**

A web application is composed by many features. These features can be connected in many different ways. As an example, to access the sales report, a user must pass through the login feature first. Another situation would be a change and/or an evolution of the login feature. As an example, imagine a login that is done only by filling up an e-mail and a password. But, the final customer asked for a captcha<sup>4</sup> on it. It means that the page will contain another HTML element to represent the captcha and the user will have to fill one more textbox.

In both cases, if the code that deals with the login page was spread among all tests that are somehow related to it, it means that the developer would have to make the change in many different places. Duplicated code is always a bad idea.

Therefore:

**Have a Page Object [Selenium, 2013] for each page of your system. The automated test code should not contain any knowledge about the pages, and must only dispatch actions on Page Objects.**

Isolating all of the code that deals with a specific page allows developers to reuse it on all automated tests. Also, it facilitates the evolution of a feature, as the only code that will be likely to change is the code that deals with that page.

Writing page objects should be a normal thing when creating system tests. In Listing 3, we show an example of a test and the Page Object that it makes use of. To make sure that the test and the Page Object contain an easy to read test, we have also made a few suggestions on how to implement it.

---

<sup>4</sup>Captcha is a test used in many web applications to determine whether the user is a human or not. More about it can be read on Wikipedia: <http://en.wikipedia.org/wiki/CAPTCHA>.

```

@Test
public void shouldGiveAnErrorMessageIfLoginIsInvalid() {
    LoginPage page = new LoginPage(driver);
    page.go();
    page.logAs("john@doe.com", "my-wrong-pass");
    Assert.assertTrue(page.errorMsgIsVisible());
}

class LoginPage extends PageObject {
    private WebDriver driver;
    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    @Override
    public void go() {
        driver.get(URLs.create("/login"));
    }

    public DashboardPage logAs(String email, String pwd) {
        driver.findElement(By.id("login")).sendKeys(email);
        driver.findElement(By.id("pwd")).sendKeys(pwd);
        driver.findElement(By.id("btnLogin")).click();

        return new DashboardPage(driver);
    }

    public boolean errorMsgIsVisible() {
        return driver.findElement(By.id("error-msg")).isDisplayed();
    }
}

```

---

Listing 3. A test, the Page Object, and the relation among them.

The test code should not know anything about the page. All navigations, actions etc, occur on the page object. The test basically dispatches and coordinates the actions to the page.

Also, all Page Objects should contain a method `go()`. Sometimes there is the need to go right to that page. This method redirects the browser to that specific page. As all pages contain an URL, a possible implementation of that would be an abstract method on the `PageObject` base class.



All asserts also happen on the test code. However, they do not extract information directly from the page. Page Objects must contain methods that return the required data to be asserted. As an example, the method `errorMsgIsVisible()` returns true if the error message has appeared on the screen.

*On Alura (Caelum, 2012), all pages in the web application have a related Page Object. In particular, the page that shows all chapters in a course changes frequently and the page object helps developers to make changes in a single place.*

## 6. Move Fast, Move Slow

*When testing a specific feature in a web application, sometimes the test navigates through many other features before or after the feature under test. Because of that, the test code can become complicated and highly coupled to all features, increasing the maintainability cost.*

\* \* \*

### **How should developers write a test that navigates through many different features before (or after) the feature under test?**

Different from unit testing, building up scenarios on a system test can be complicated. One way to build those scenarios is navigating on many features and filling the application up with the required data. Also, sometimes the web application requires the user to navigate through many pages before accessing the main feature.

Imagine a web application that stores a task list. To access this list, users should log in first. Therefore, there are two different test suites to this application: the one that tests the log in and the one that tests the task list itself. In Listing 4, we present a test that guarantees that an error occurs if the user makes a mistake. One can see that the test is very detailed in terms of how the functionality should work: fill the form, check the checkbox to save the password, fill the captcha and then submit the form.

---

```
@Test
public void shouldGiveAnErrorMessageIfLoginIsInvalid() {
    LoginPage login = new LoginPage(driver);
    login.fillForm("john@doe.com", "123");
    login.checkSavePassword();
    login.captcha("word");
    login.submit();
    assertTrue(page.errorMessageIsVisible());
}
```

---

Listing 4. Testing the login feature.

Now, imagine the automated test of the task list feature. It will contain all the lines from Listing 3 (the test needs to log in first), plus the code that will validate the feature under test. That may not be a good idea for a few reasons: (i) the code will be very long, depending on the number of features the test has to navigate before testing the feature under test, (ii) if something changes on the feature, the change will have to be propagated on many different test suites.

Therefore:

**When writing the test code that navigates on the feature under test, the test code must go through every single step explicitly, highlighting each detail in that feature; when navigating on any other feature (at that time, not under test), the test must skip details, and just execute the required action.**

In this case, the Login test suite should make use of all specific methods that exists in the Login Page Object. However, this Page Object should also have a method that encapsulates all of the process of login. This method should be used by other test suites. In Listing 5, we show an example of the test of task list. One can notice that the `doLogin()` method does the full process of logging in.

If the feature changes for some reason, then the developer should update the Login test suite (which is already expected) and the `doLogin()` method. All test suites will keep working.

---

```
@Test
public void shouldAddAnItemOnTheTaskList() {
    LoginPage login = new LoginPage(driver);
    login.doLogin("john@doe.com", "123");

    TaskListPage taskList = new TaskListPage(driver);
    taskList.go();
    taskList.add("Do my homework");
    taskList.submit();

    assertTrue(taskList.addMessageIsVisible());
}
```

---

Listing 5. Testing the task list feature.

*On Alura (Caelum, 2012), there are methods that encapsulate the use of a full feature. They only*

*exist to facilitate the writing of the test for a feature that requires the use of other features.*

## 7. Never Look at the HTML Source

*Assertions in web testing applications commonly check if some message/element has appeared on the HTML. However, when searching for a message in the HTML, the test may pass even if the message is not on the place it is meant to be.*

\* \* \*

### **How should the test validate the existence of a text in a HTML page?**

All automated tests contain asserts. Asserts are the way to guarantee that your software worked as expected. Assertion in unit tests usually check if the object is on a valid state, with the expected data on its fields. On the other hand, asserts in a web test are different: the test should verify if a message or even an element on the HTML contains the expected content (or has the expected CSS class, and so on).

Suppose that the login feature should exhibit a message if the authentication fails: *"User or password is invalid"*. A possible assert for that would be to check the existence of this text in the whole HTML code. This is very easy to be done as all web testing frameworks give access to the full HTML.

However, this approach can be dangerous. Depending on what the assert is looking for, it may find the same text, but on the wrong place. In that case, the test will pass instead of failing.

Therefore:

**Assert the presence of a text on its specific place/element. Never look for the existence of a text in the full HTML source.**

In Listing 6, we show a good and a bad example of asserting the presence of a message. Also, depending on the framework, the behavior of the method that returns the full HTML can vary. Selenium<sup>5</sup>, for example, returns the current HTML even if it is not entirely loaded yet. Because of that, your test may fail not because the message did not appear, but because the HTML is not yet complete.

---

<sup>5</sup> <http://code.google.com/p/selenium/>. Last access on May, the 11th, 2014.

```
_____  
// wrong way  
assertTrue(driver.getPageSource().contains("User or password invalid"));  
  
// right way  
assertEquals("User os password invalid",  
driver.findElement(By.id("msg-box")).getText());
```

\_\_\_\_\_  
Listing 6. Asserting the presence of a text on its specific place.

*On Alura (Caelum, 2012), there were many asserts directly on the HTML page source. The tests were highly unstable, as sometimes the `getPageSource()` did not work return the full HTML. Nowadays, all validations are done at their specific places.*

## 8. Conclusion

Writing automated web tests can be tricky. As it demands a lot of source code, it is easy to duplicate code and create highly unstable tests. In this paper, we have presented a set of patterns that helps developers to maintain the internal quality of their test suites.

Still, as we have noticed in our case study, even after using all these patterns, the test suite still contains a few problems that need to be solved. In a future work, the solutions that the team is about to create, may become new patterns.

## ACKNOWLEDGEMENT

We thank Caelum Ensino e Inovação (<http://www.caelum.com.br>) for supporting this research. In particular, the members of team: Caio Incau, Caio Souza, Francisco Sokol, Leonardo Wolter, Suellen Goulart, and Guilherme Silveira.

## REFERENCES

Caelum, Alura. E-learning system. <http://www.alura.com.br>. Last access on May, the 11th, 2014. 2012.

Freeman, S. and Mackinnon, T. and Pryce, N. and Walnes, J. 2004. Mock roles, not objects, In: OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM, p. 236-246.

Janzen, D., Software Architecture Improvement through Test-Driven Development. Conference

on Object Oriented Programming Systems Languages and Applications, ACM, 2005.

Maximilien, E. M. and L. Williams. Assessing test-driven development at IBM. IEEE 25th International Conference on Software Engineering, Portland, Orlando, USA, IEEE Computer Society, 2003.

Nagappan, N., Bhat, T. Evaluating the efficacy of test- driven development: industrial case studies. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering.

Selenium Documentation. Page Objects. <https://code.google.com/p/selenium/wiki/PageObjects>. Last access on May, the 11th, 2014. 2013.