Software Engineering — Test Automation

Arkadiusz Chrobot

Department of Information Systems, Kielce University of Technology

Kielce, January 6, 2025

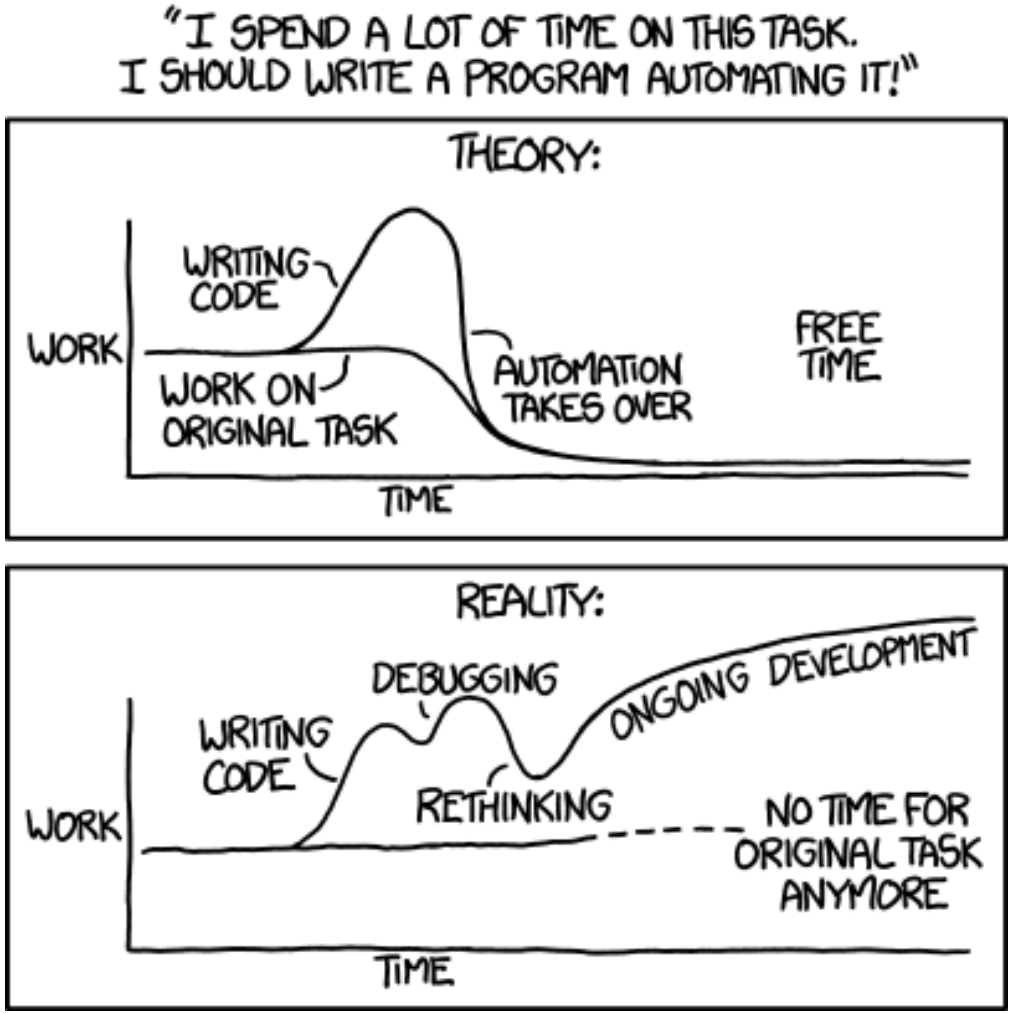
Outline

Motto

”Why program by hand in five days what you can spend five years of your life automating?”

— Terence Parr

Motto



Source: xkcd.com

Introduction

Test Automation not only reduces the costs of the *Quality Control*, but also allows software engineers to monitor the progress of their work and allows them to safely (to some extent) introduce modifications to the code. Automated tests are also a form of *software documentation*. The automation makes also the tests repeatable. However, to bring these benefits, the test automation has to be done the right way.
In this lecture, we are going to discuss the principles and tools that allow software engineers to automate the unit, integration and system tests.

Unit Testing

F.I.R.S.T Principles

Automated unit (and also integration) tests should follow the F.I.R.S.T principles:

Fast Tests should be fast. They are supposed to provide feedback to software engineers as soon as possible. If they are slow, then programmers tend to avoid running them, and that causes defects to stay undetected in the code for a long time.

Independent No test should depend on the outcomes of other tests. They should also not depend on the order in which they are run. Otherwise, a failure of one of the tests will cause a cascade of failures in next tests.

Repeatable It should be possible to run tests in any reasonable environment. A software engineer should be able to perform tests on her or his laptop, as well as in a CI/CD pipeline.

Unit Testing

F.I.R.S.T Principles — Continued

Self-Validating The result of a test should be an information, whether it has passed or failed. The cause of the failure is an important piece of information, but it shouldn't obscure the outcome of a test. A software engineer shouldn't be forced to read through long logs in order to find out whether the test was successful, or not.

Timely Tests should be created before the production code they are intended to verify. Otherwise, the code may turn out to be difficult or even impossible to test.

Unit Testing

Test-Driven Development

The last of the F.I.R.S.T rules summarizes the *Test-Driven Development* (*TDD*) methodology of creating software, whose key principles are:

1. Write unit tests *before* the production code.
2. If a unit test fails, then write production code only to pass this test.
3. Don't write more unit tests.
4. Write only as much production code as it is required for passing single failing test.

The TDD has probably as many proponents as opponents. The latter point out, that focussing too much on passing tests may result in poor software architecture and usability, if the test cases are chosen incorrectly. If programmers don't follow the 3^{rd} principle, then they may create tests that block any possibility of modifying the code.
The TDD approach is certainly useful, but it should be applied with care.

Unit Testing

Tools

Many libraries and frameworks have been developed, that support unit and integration testing. The most popular in Java ecosystem is the *JUnit* library, that currently is available in its fifth version. An interesting alternative to JUnit is TestNG, however JUnit is part of a larger family of unit test libraries, commonly referred to as *x*Unit. The first member of this family is the SUnit, developed for the programming language Smalltalk.
The JUnit, like other unit test libraries, allows software engineers to separate the test code from the production code. It means that tests are not by default deployed to the production environment. Moreover, JUnit is supported by popular build automation tools, like Maven or Gradle.
Tests in JUnit are implemented in separate classes. The name of the class is usually derived from the name of the class that its tests verify, but *must* end, or begin, with the word *Test* or *Tests*. Each test is implemented as a separate method, marked with an appropriate *annotation*. It is recommended that such a method should contain only one *assertion*, but sometimes it is necessary to add more than one.

Unit Testing

Annotations

JUnit 5 provides many annotations that make it possible to create tests. Some of them are listed in the table 1.

Table: JUnit 5 Annotations

Annotation	**Description**
@Test	Marks a method as a test method.
@ParameterizedTest	Denotes that a method is a parametrized test.
@RepeatedTest	Allows the test to be repeated.
@TestMethodOrder	Defines the order of test methods execution.
@Disabled	Disables the test.
@Timeout	Specifies the timeout for a test.
@BeforeEach	The method is performed before every test.
@AfterEach	The method is performed after every test.
@BeforeAll	The method is performed before all tests.
@AfterAll	The method is performed after all tests.

Unit Testing

Annotations

The parametrized test method requires another annotation that specifies its source of input data and expected results. This annotation is `@MethodSource`. The argument for this annotation is the name of a static method that provides the needed data. That method should return a stream of objects of the `Arguments` class.
The `@RepeatedTest` annotation also requires an argument, which is the number of test repetitions.

Unit Testing

Assertions

Assertions are conditions that should always be met, if the code under verification works correctly. In JUnit terminology an assertion is a static method of the `Assertions` class, that checks if the test has passed, or not. Some of them are described in the table 2. There are many overloaded versions of those methods, for different data types of `expected` and `actual` parameters. Some of them accept an additional parameter, that is a message displayed, when a test fails. The JUnit makes it possible to use assertions from other libraries or frameworks, like Hamcrest.

Unit Testing

Assertions

Table: JUnit 5 Annotations

Assertion	**Description**
`assertArrayEquals(expected, actual)`	Asserts that the `expected` and `actual` arrays contain the same values.
`asertEquals(expected, actual)`	Asserts that `expected` value is equal to `actual`.
`assertNotEquals(expected,actual)`	The opposite of the above assertion.
`assertTrue(condition)`	Asserts that the `condition` is true.
`assertFalse(condition)`	Asserts that the `condition` is false.
`assertNull(reference)`	Asserts that the `reference` is `null`.
`assertNotNull(reference)`	Asserts that the `reference` is not `null`.

Mocks

The automated integration tests may require test doubles. There are several frameworks that help to create mocks and other such objects. For Java one of them is EasyMock. In this lecture however, another such a framework is described, Mockito.

To make the code easy to test, software engineers often apply the *Dependency Injection* design pattern. In this pattern an object doesn't create internally objects it needs, instead it receives them usually by constructor parameters. This allows programmers to apply mocks easier.

The test class that needs to apply mocks form the Mockito framework needs to be annotated with the `@ExtendWith` annotation. The argument for this annotation should be `MockitoExtension.class`. The reference to the mocked object should be marked with the `@Mock` annotation. The method that uses this reference should check if it is not `null`. The behaviour of the mock may be specified with the use of the `when(condition).thenReturn(result)` statement.

Mocks

The Mockito framework provides also spies. A spy is an object that wraps up another object, and verifies how it behaves under tests, for example how many times its methods are invoked. The spy can be created with the use of the `@Spy` annotation or by passing an actual object to the static `spy()` method. To verify how many times a given method of the spied object has been invoked the following statement can be applied:

```
verify(spyReference, atLeastOnce()).methodName(arguments)
```

Instead of `atLeastOnce()` several other methods for counting invocations may be applied, like `never()`, `atMostOnce()`.

End-To-End Testing

The *End-To-End* (*E2E*) Tests are form of functional tests, that are performed on the system testing level. They are often performed manually for the first time, but then they can be automated. There are several tools that make it possible. Choosing them depends on the verified software user interface. If it is text-based, then simple input and output redirection and some shell scripting is enough. In case of applications with GUI, tools may be used that record the interaction with user and then replay it. Some of them can convert the recording into a script, which then may be modified. An example of such a tool is QF-TEST. Unfortunately, these tools have some limitations, that may exclude them from using in particular situations. In such cases the automated test can be created manually with the use of special libraries. In Java for the AWT and Swing-based GUIs are available Jemmy and Jemmy 2 libraries. For JavaFX-based GUIs, a better choice is the TestFX library.

End-To-End Testing

TestFX Library

The TestFX library allows the testers to create a (virtual) *robot*, that clicks buttons, populates text fields, selects options from menu, in short, uses the GUI. Although the library provides its own assertion, it requires support from testing library, like JUnit, to be able to perform tests. The class that implements tests should inherit the TestFX `ApplicationTest` class. The test methods can get references to elements of the GUI, using CSS selectors for attributes like classes, IDs and others. The method responsible for locating these elements is `lookpu()`. It returns an object that has the `querryAll()` method, that returns a collection of elements that have the required attribute. A single object from this collection can be acquired with the use of the `next()` method returned by the `iterator()` method. An element can be clicked with the use of `clickOn()` method.
The TestFX library has its own set of *matchers*, i.e. objects that check if a given condition is met. For example the `hasText()` method of the `LabeledMatchers` class checks whether a given GUI element has the required description. There is also the `FXAssert` class that provides assertions methods, like `verifyThat()`.

End-To-End Testing

Automated Tests for Web Applications

Tests of Web Applications may be automated with the use of tools like QF-TEST, but there are also tools that allow the software engineers to create automated tests manually. Nowadays, all of them use the WebDriver protocol that allows them to interact with, and control a web browser. All modern browsers support this API, and their producers provide necessary drivers. Among these tools are frameworks like Cypress and libraries like Selenide. The oldest and probably the most popular of them is Selenium.

End-To-End Testing

Automated Tests for Web Applications

The most frequently used design pattern in software that implements web application tests is the *PageObject* pattern [1]. It requires creating objects that represent and control web pages of the tested application. The Figure 1 describes this concept.

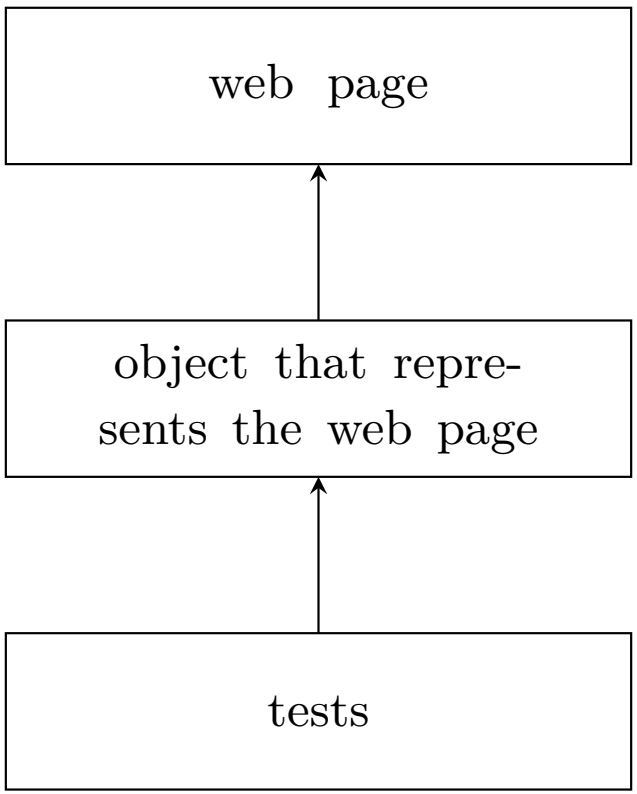


Figure: The PageObject Design Pattern

End-To-End Testing

Automated Tests for Web Applications — Selenium

The most important class of the Selenium library is the `WebDriver`, that allows the test to get access to and control web page elements. The elements may be retrieved with the use of the `findElement()` or `findElements()` methods. Several types of *locators* may be applied, the most popular being *XPath*. Others include CSS Selectors, IDs, names, class names, tag names, links, and partial link texts. The `get()` method of the `WebDriver` class orders the web browser to load a specified web page.
A single element of the web page is represented by an object of the `WebElement` class. Selenium allows for implicit and explicit waiting for a web page element to become available. The implicit waiting period is defined globally. The explicit waiting strategies are provided by the `WebDriverWait` and `FulentWait` classes. The latter is the most flexible.

Bibliography

[1] Unmesh Gundecha. *Selenium Testing Tools Cookbook*. Second Edition. Birmingham, UK: Packt Publishing, 2015.

Questions

?

The End

Thank You for Your attention!