Recent Trends in Software Testing – A Case Study with Google Calendar

BOJAN POPOV, BOJANA KOTESKA and ANASTAS MISHEV, Ss. Cyril and Methodius University

In this paper we make an overview of the software testing trends and as a case study we perform testing of the Google Calendar service. We present some of the latest testing techniques, frameworks and tools used for commercial software. Finally, we perform black-box automated testing of the Google Calendar component by applying several different testing technologies and frameworks. We use JUnit, Selenium and Mockito frameworks to create 22 tests to perform interface and functionality testing.

1. INTRODUCTION

Testing is one of the most important and crucial parts in developing any kind of software [Ammann and Offutt 2016]. Approximately half budget spent on the development of the software projects is spent on software testing [Harman et al. 2015]. To create quality software (reliable, secure, etc.) means to test the software from different aspects and to create optimal combination of different coverage criteria because of the deadlines and system size [Rojas et al. 2015]. The purpose of the test team is to provide effectively and efficiently accurate and useful testing services and quality information about the project [Black 2016].

One of the common software testing categorization is: black box testing, white box testing and gray box (combination of black box and white box) [Jan et al. 2016]. If the software testers/developers have the software source code then they can use both black box and white box testing, but if they do not have the internal source code, then, only a black box testing is possible. Usually, black box testing, also called a functional testing, is performed by using the software documentation and design specification [Nidhra and Dondeti 2012].

From another point of view, the testing can be divided into automated and manual testing. In a case of manual testing, the software testers have the role of end users and they are responsible for checking the behavior of software by following a test plan and a set of test cases. Automated testing is performed by using a specific testing software framework that executes the tests and compares the actual outputs with the expected outputs [Garousi and Mäntylä 2016].

Software testing includes different level of testing such as unit, integration, system, and acceptance testing. Unit and integration testing focus on individual modules, while system and acceptance testing focus on overall behavior of the system [Dhir and Kumar 2019].

This work is supported by the Faculty of Computer Science and Engineering, Skopje, North Macedonia.
Bojan Popov, Bojana Koteska, Anastas Mishev, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: bojan.popov@students.finki.ukim.mk; bojana.koteska@finki.ukim.mk; anastas.mishev@finki.ukim.mk.

In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

In this paper we make an overview of the current trends in software testing, including methods, practices and frameworks. As a case study we perform black-box testing of the Google Calendar service [Google 2006] by using different testing technologies and frameworks (JUnit, Selenium and Mockito).

The paper is organized as follows. Section 2 gives an overview of the current trends in software testing. Section 3 provides a comprehensive explanation of the testing methodology, definition of test cases, test generation and design the results. The final Section 4 concludes the paper.

2. RECENT TRENDS IN SOFTWARE TESTING

Agile testing is a trend in software testing which follows the rules of agile development policy. It considers software improvement as a critical part like a client in the testing process. In agile development the code is written during each iteration and the testing is done after each iteration. In [Dhir and Kumar 2019], the authors propose a model for automated agile testing and as an experiment they perform testing by using the Selenium tool [Jason Huggins 2004] which is an automating web application testing framework. Selenium is probably the most widely-used open source solution for testing web applications [Gundecha 2012]. The Selenium WebDriver provides opportunity to create custom automation framework that could reduce development time, increase the return of investments and to minimize the risks [Vila et al. 2017]. Another tool interesting for the Agile market is Testsigma [inc 2019]. It is one of the best automation tools which is AI-driven and automates complex tests using simple English and no programming.

Running unit tests in parallel and distributed environments can significantly speed up the time required for test execution. A tool for automatic execution in distributed environments CUT (Cloud Unit Testing) is presented in [Gambi et al. 2017]. Based on the set of unit tests, this program allocates computational resources such as virtual machines or containers and schedules the execution of the unit tests over them. The tool is implemented in Java and it monitors its progress directly inside the JUnit [The JUnit Team 2017] test automation framework which from the developers' point of view the execution of the tests looks same like they are executing locally. JUnit platform is used as a foundation for launching tests on the Java Virtual Machine (JVM).

Automated program repair (APR) techniques became very popular in the recent years [Wang et al. 2019]. They have shown to be promising in increasing the effectiveness of automated debugging [Adamsen et al. 2017]. The general idea of these techniques is to provide automatic software repair and to produce fix of the programs which needs to be validated by software testers. The benefit of these techniques is that they fix the software and they decrease the effort to identify and correct faults [Gazzola et al. 2017].

The elimination of unnecessary test cases or selection of specific test cases can significantly reduce the time for testing. The goal of the test suite minimization is to distinguish repetitive experiments and to eliminate the redundant test cases. Test case selection decreases the number of test cases to be executed by recognizing the important test cases (test cases related to the latest changes of the software). Lately, test case prioritization is becoming trend in software testing the testing is performed by an early optimization based on proffered properties. It means that the test cases that are considered as highly significant will be executed first and will provide feedback to the testers earlier [Khatibsyarbini et al. 2018]. In [Azizi and Do 2018], the authors propose an item-based collaborative filtering recommending system that uses user interaction data and application change history information to develop a test case prioritization technique. A multi-objective optimisation technique is used to analyse the trade-off between the code coverage and execution time of the test suite written by programmers who are experts in testing [Turner et al. 2016].

Testing of an object-oriented code can be difficult because of the dependency objects. When dependency objects occur, then software testers can mock objects and simulate the complex dependency.

Mocking is a common approach in unit testing which isolates a class from its dependencies by replacing a dependency class instead of the original one. Authors in [Arcuri et al. 2017] create mock objects using the popular Mockito framework [Faber 2008]. Mockito framework is used for effective unit testing of JAVA applications. It is used to mock interfaces so that a dummy functionality can be added to a mock interface. It can simulate a method call and return result as needed. It is a very powerful tool for mocking database objects, and some functionalities that require time and acceptance from other users of the system. Mockito is among the top 10 most used Java libraries hosted on on GitHub [Arcuri et al. 2017].

3. TESTING OF THE GOOGLE CALENDAR SERVICE

Different testing methods have been found in the literature for testing Google services. For example, the Google Map search application was tested by performing an improved random testing using some predefined values also used for verifying specific properties. The framework used for testing was composed of several testers which control and monitor the test execution [Salva and Laurencot 2009]. Brown et al. [Brown et al. 2018] propose a metamorphic testing strategy to test the Google Maps mobile app navigation. They tested its web service API and its graphical user interface and detected several real-life bugs in the Google Maps app.

In [Carlini et al. 2012], the authors analyzed 100 Google Chrome extensions, including the 50 most popular extensions to determine whether Chrome's security mechanisms successfully prevent or mitigate extension vulnerabilities. They find that 40 extensions contain at least one type of vulnerability. The testing was performed by using a three-step security review process: black box testing, source code analysis and holistic testing.

3.1 Definition of Test Cases

Google Calendar provide multiple options: to create public events and share with other people; to create private events; to create default calendars or to add calendars created by other users; to delete events; to change date, time and color of the specific event, etc. We test the basic functionalities and some of the additional functionalities that Google Calendar has. The purpose of the testing is to find the suitable testing frameworks and to check if both the interface and functionalities of Google Calendar service work properly.

The following test cases were defined as a part of the interface testing:

(1) Counting events in a specific calendar view;

(2) Change the view of the calendar and count the days that are displayed depending on the view;

(3) Deactivate and activate additional calendars (for example: display of holidays - religious, national);

(4) Click the back and forward buttons to change day/week/month/year.

The functional testing covers the following test scenarios:

(1) Create an event;

(2) Search events;

(3) Delete event;

(4) User login;

(5) User logout;

(6) Edit event (change time, color, date, etc.)

3.2 Testing Frameworks

To perform the testing of the Google Calendar we use three different testing frameworks. All tests are written in Java 8, JavaSE-1.8 [Oracle 2014].

The core framework we use for testing is JUnit. Next, we use Selenium to test the web pages and the last one is Mockito, which simulates responses of some Google Calendar functionalities. The capabilities of these frameworks are described in Section 2.

3.2.1 *JUnit.* Testing of the Google Calendar service is made by using JUnit5 which requires minimum Java version 8. All methods annotated with **@Test** are considered as tests in JUnit.

Figure 1 represents a simple test written in JUnit 5. The purpose of this test is to check the sign-in button click on the Google home page (www.google.com). After that, a new page should be loaded which provides user login.

```
@Test
public void testaa() throws InterruptedException {
    String[] return_login_string_split  = c.clickLogin().split("\n");
    assertTrue((return_login_string_split[0] + " " + return_login_string_split[1])
            .equals("Sign in with your Google Account"));
}
```

Fig. 1. JUnit test

3.2.2 *Selenium.* A Selenium WebDriver API and Firefox web driver were used to perform testing of the Google Calendar web pages. Selenium WebDriver object provides access to an HTML page. Some of the basic functions of Selenium can be seen in Figure 2. The sleep method allows specific waiting time for the page loading. The web driver object allows users to access the HTML page web elements, but it is important to specify the XPath or ID of the web element.

```
public String clickLogin() throws InterruptedException {
    Thread.sleep(1500);
    driver.findElement(By.id("gb_70")).click();
    Thread.sleep(1500);
    return driver.findElement(By.xpath("//div[@jsname=\"tJHJj\"]")).getText().trim();
}
```

Fig. 2. Using Selenium WebDriver to find specific HTML web element

3.2.3 *Mockito.* Figure 3 is an example of how Mockito works with JUnit. A Dao class (DaoCalendar) is mocked and the mocked object is passed to the Dao service class (DaoServiceCalendar).

3.3 Test Creation and Execution

A total of 22 tests were defined and each functionality was tested separately. We organized the tests so that the login test is first. Second, we test the calendar functionalities, and log out process is tested last. The goal of the first login test is to click the login button on the Google home page, then to enter the email address and password and finally to click the sign-in button. These tests are shown in Fig. 4.

The first test (*testaa*) tests if the page loads correctly after clicking the sign-in button.

```java
@Test
public void testea() throws InterruptedException {
    dao = Mockito.mock(DAOCalendar.class);
    DAOServiceCalendar daoService = new DAOServiceCalendar(dao);
    DAOServiceCalendar spy = Mockito.spy(daoService);
    Mockito.doNothing().when(spy).addCalendar(Mockito.isA(String.class));
        Mockito.when(dao.returnNumberOfCalendars()).thenReturn(6);
    dao.addCalendar("somebody@email.com");
    assertEquals(6, dao.returnNumberOfCalendars());
}
```

Fig. 3. Mockito usage in JUnit test

```java
@Test
public void testaa() throws InterruptedException {
    String[] return_login_string_split  = c.clickLogin().split("\n");
    assertTrue((return_login_string_split[0] + " " + return_login_string_split[1])
            .equals("Sign in with your Google Account"));
}

@Test
public void testab() throws InterruptedException {
    String email = "your_email_here!";
    String string = c.enterEmailAddress(email);
    String[] return_password_string_split = string.split("\n");
    assertTrue((return_password_string_split[0] + " " + return_password_string_split[1]).equals("Welcome "+email));
}

@Test
public void testac() throws InterruptedException {
    String password = "your_password_here";
    assertTrue(c.enterPassword(password).isEnabled());
}
```

Fig. 4. Login tests

```java
 * function that clicks on the login login button
public String clickLogin() throws InterruptedException {
    Thread.sleep(1500);
    driver.findElement(By.id("gb_70")).click();
    Thread.sleep(1500);
    return driver.findElement(By.xpath("//div[@jsname=\"tJHJj\"]")).getText().trim();
}
```

Fig. 5. Clicking login button

The function **clickLogin** from the test *testaa* is shown in Fig. 5. Three seconds are given for loading (1500 ms for loading the Google home page and extra 1500 ms after clicking the sign-in button), from which we take the text that gives feedback of the entered sign-in page.

Figure 6 provides details about **enterEmailAddress** (*testab*) and **enterPassword** (*testac*) functions.

The test *testab* first enters the email address by calling the **enterEmailAddress** function which clicks the ”next” button and gets the text for the password form.

```
 * function that enters the given email into the web page field for email
public String enterEmailAddress(String email) throws InterruptedException {
    driver.findElement(By.id("identifierId")).sendKeys(email);
    Thread.sleep(1500);
    driver.findElement(By.id("identifierNext")).click();
    Thread.sleep(1500);
    return driver.findElement(By.xpath("//div[@jsname=\"tJHJj\"]")).getText();
}

 * function that enters the password in the web page field for password
public WebElement enterPassword(String password) throws InterruptedException{
    driver.findElement(By.xpath("//input[@jsname=\"YPqjbf\"]")).sendKeys(password);
    Thread.sleep(1500);
    driver.findElement(By.id("passwordNext")).click();
    Thread.sleep(1100);
    return driver.findElement(By.xpath("//a[@title=\"Google Account: Bojan Popov  \n" +
            "(bojanpopovred@gmail.com)\"]"));
}
```

Fig. 6. Entering email and password functions

The test *testac* calls the method **enterPassword** which enters the password, clicks the ”sign-in” button and gets the text for successful login.

After the login, the URL is changed to the URL of Google Calendar, from where the calendar functionality testing starts.

To test change of a view, we created parameterized tests that provide testing with multiple inputs. There are several options for each view and each of these options was tested separately. The simulation of button clicks and the click responses were also considered in multiple tests and their goal was to detect a page change.

In order to test the creation of an event, the number of existing events in the calendar was checked, and after a new event was created, this number was checked again. The tests for this part are shown in Fig. 7.

The first test *testbf* gets the number of existing events before creating a new event. The goal in this test is to click on the calendar form for creating events and click on more details. Then, the second test *testbg* enters the event parameters (date, start time, end time, name) and it checks the number of notifications about the new event. If everything is performed correctly, the result should be 1.

The last test *testbh* enters the additional event parameters such as color, it clicks the ”save” button and checks the number of events again.

The delete event functionality is tested by using the search bar provided by Google Calendar. Fig. 8 shows the test for deleting an event. It counts the number of events before and after the event is deleted. The **removeEvent** function first searches the event, gets all the events with the searched name, clicks the first event and then deletes the event by clicking the ”delete event” button.

Furthermore, there are functionalities in Google Calendar that wait for a certain response and these parts were tested with Mockito framework. We simulated that a certain response was sent, but in the meantime, we tested if the button was clickable.

The test file was written in a way so that the tests were defined as simple method calls and JUnit methods (assertTrue or assertFalse) were used for the evaluation of the correctness of the results.

```
@Test
public void testbf() throws InterruptedException {
    numberOfEventOnCalendar = c.getNumberOfEvents();
    assertEquals("Add title", c.openingCreateEvent());
}

@Test
public void testbg() throws InterruptedException {
    c.enterParameters("9 June 2019","2:30pm","9:35pm","My free time");
    assertEquals(1,c.numberOfNotifications());
}

@Test
public void testbh() throws InterruptedException {
    c.restParametersSettings();
    assertEquals(c.getNumberOfEvents() - 1, numberOfEventOnCalendar);
}
```

Fig. 7. Creating event tests

```
@Test
public void testca() throws InterruptedException {
    numberOfEventOnCalendar = c.getNumberOfEvents();
    c.removeEvent("My free time");
    assertEquals(c.getNumberOfEvents(), numberOfEventOnCalendar - 1);
}
```

Fig. 8. Delete event test cases

The test class contains a service class which is responsible for the button clicks, for the searching through the web page elements, for creating new events, deleting events, etc.

There is also one more service class which is responsible for changing the dates. Since each month has a specific number of days we used the Java Calendar class which returns the number of days in a not leap year.

Table I shows the results of the executed tests. As it is shown, the only test that failed is ”Testing calendar days after changing a view”. Errors occurred in such occasions, mostly in the part of deleting an already created event. Sometimes the event was not deleted from the calendar, but when that test was executed no failure was detected. The conclusion after a couple of tests executed was that Google Calendar does not load fast and some of the page elements are not fully loaded at the time of the test execution which was the case with the failed test also.

Table I. Results of the executed tests

Tests	**Result**
Log in	pass
Testing calendar days after change of views	failure
Creating an event	pass
Editing an event	pass
Deleting an event	pass
Log out	pass

It is important to mention that Google constantly releases new updates of Google Calendar with changed interface. It means that HTML elements, which are important in Selenium, have different IDs and class names. The interface changes caused some of the tests to fail and we had to change them multiple times. Table II shows the execution results after the Google calendar update.

Table II. Results of the executed tests after the Google Calendar update

Tests	**Result**
Log in	pass
Changing view	error
Testing calendar days after change of views	failure
Creating an event	pass
Editing an event	pass
Deleting an event	pass
Log out	pass

4. CONCLUSION

In this paper we provided an overview of some of the new technologies and practices in software testing and we applied them in practice by making black-box testing of the Google Calendar component. The testing was made by using some of the popular testing frameworks such as JUnit, Selenium and Mockito. A total of 22 tests were created to test the interface and the functionalities of the Google Calendar.

In conclusion, we believe that good software testing requires a lot of practice and constant monitoring of world trends in the field of software engineering. Testers should make a test strategy and find a balance between code coverage and time required for testing. There are many useful testing tools that can automate and ease the process of testing. Some of them do not even require programming skills.

From the testing process of the Google Calendar service, we learned that an integration of more testing frameworks and test automation can be done fast and with small effort, but it requires field knowledge. JUnit provides user friendly way to automate the test execution by creating test suites. The challenging task is to define the test scenarios and to find which tools and techniques are more suitable.

REFERENCES

Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing event race errors by controlling nondeterminism. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 289–299.

Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private api access and functional mocking in automated unit test generation. In *2017 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 126–137.

Maral Azizi and Hyunsook Do. 2018. A collaborative filtering recommender system for test case prioritization in web applications. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, 1560–1567.

Rex Black. 2016. *Pragmatic software testing: Becoming an effective and efficient test professional*. John Wiley & Sons.

Joshua Brown, Zhi Quan Zhou, and Yang-Wai Chow. 2018. Metamorphic testing of navigation software: A pilot study with Google Maps. In *Proceedings of the 51st Hawaii International Conference on System Sciences*.

Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An evaluation of the google chrome extension security architecture. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 97–111.

Saru Dhir and Deepak Kumar. 2019. Automation Software Testing on Web-Based Application. In *Software Engineering*. Springer, 691–698.

Szczepan Faber. 2008. Mockito. (2008). https://site.mockito.org/

Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. 2017. CUT: automatic unit testing in the cloud. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 364–367.

Vahid Garousi and Mika V Mäntylä. 2016. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology* 76 (2016), 92–117.

Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

Google. 2006. Google Calendar. (2006). https://www.google.com/calendar

Unmesh Gundecha. 2012. *Selenium Testing Tools Cookbook*. Packt Publishing Ltd.

Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–12.

Testsigma inc. 2019. Testsigma. (2019). https://testsigma.com

Syed Roohullah Jan, Syed Tauhid Ullah Shah, Zia Ullah Johar, Yasin Shah, and Fazlullah Khan. 2016. An innovative approach to investigate various software testing techniques and strategies. *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN* (2016), 2395–1990.

Jason Huggins. 2004. JUnit 5. (2004). https://www.seleniumhq.org/

Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93.

Srinivas Nidhra and Jagruthi Dondeti. 2012. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)* 2, 2 (2012), 29–50.

Oracle. 2014. Java 8. (2014). https://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html

José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93–108.

Sébastien Salva and Patrice Laurencot. 2009. Automatic Ajax application testing. In *2009 Fourth International Conference on Internet and Web Applications and Services*. IEEE, 229–234.

The JUnit Team. 2017. JUnit 5. (2017). https://junit.org/junit5/

Andrew J Turner, David R White, and John H Drake. 2016. Multi-objective Regression Test Suite Minimisation for Mockito. In *International Symposium on Search Based Software Engineering*. Springer, 244–249.

Elior Vila, Galia Novakova, and Diana Todorova. 2017. Automation testing framework for web applications with Selenium WebDriver: Opportunities and threats. In *Proceedings of the International Conference on Advances in Image Processing*. ACM, 144–150.

Shangwen Wang, Ming Wen, Xiaoguang Mao, and Deheng Yang. 2019. Attention please: Consider Mockito when evaluating newly proposed automated program repair techniques. In *Proceedings of the Evaluation and Assessment on Software Engineering*. ACM, 260–266.