# Collecting Code Coverage from UI Testing

1st Filip Gurbáľ
*Department of Computers and Informatics*
*Faculty of Electrical Engineering and Informatics*
*Technical University of Košice*
Košice, Slovakia
filip.gurbal@tuke.sk

2nd Jaroslav Porubän
*Department of Computers and Informatics*
*Faculty of Electrical Engineering and Informatics*
*Technical University of Košice*
Košice, Slovakia
jaroslav.poruban@tuke.sk

***Abstract*—When testers design test cases based only on specification, they can introduce significant redundancies into them and also create gaps of untested software. More insight into the impact of test cases on implementation can guide testers to design more effective test cases to cover implementation and prevent redundancies to save time. In this paper we describe challenges of UI tester and how we can help them using coverage analysis of UI source code. We prepared infrastructure to automatically collect code coverage data from UI testing for further analysis.**

***Index Terms*—code coverage, manual testing, UI testing**

## I. Introduction

Assuring quality of the software is demanding task but it is important part of software development. It takes care of testing of the product to confirm its correctness and reliability. Improper testing can fail to detect critical software failures causing huge losses. It is advised to test software earlier in the development [1], because finding and fixing faults later can be significantly more expensive [2].

Estimations say that 80% of software development cost is spent on detecting and fixing defects [3]. This is why researchers try to improve testing techniques, methods and tools. We know multiple types of testing based on the size of the software part that is subjected to testing. Hooda and Chhillar's [4] summarized main types of functional testing, from which we selected three that are performed by testers during the development:

- *Unit testing.* The lowest level of testing mainly performed by developer to test the unit of code.
- *Integration testing.* Focuses on communication between different modules of the system.
- *System testing.* Testing of the overall system deployed in the target environment. This type of testing includes testing of user interfaces (UI testing) and end-to-end testing (E2E testing).

If testing is systematic and testers document their test cases, certainly the most complex testing is the system testing. For instance, in case of web application the tester tries to design user scenarios that will reliably validate the functionality against the specification. User scenarios are various ways how a user can use the software. Ideally the tester will find every possible use case of the software and will manually try it to observe the behavior. If anything changes in behavior or user interface, every affected scenario need to be retested. The tester will invest too much time to find as many use cases as possible, document them into test cases, try them and then repeat the process after any change in the software or specification.

In this example the tester designed test cases only based on the specification. Richardson, O'Malley and Tittle [5] identified two problems with this approach:

- significant redundancies may be introduced into test cases,
- test cases based only on specification can create big gaps of untested software.

The problems are caused by the big gap between the test case and the source code as shown in figure **??**. The tester may design many test cases that in fact execute the same part of the source code, in which case the test cases are redundant or ineffective. Also there can be lot of source code for functionalities not mentioned in specification.

Some of the time can be saved by automating the test cases, so the tester does not need to test manually every time something change. Contan, Dehelean and Miclea [6] stated that automated UI testing (which falls under the system testing, because it requires all modules to be integrated and working) have the following negative attributes:

- UI tests are fragile, meaning that any small change in the software can break many test cases, and
- UI tests are time consuming,

This is why we also need to know what test should be selected for automation and which automated test cases should be included in regression testing. In our research we want to address following UI testing challenges:

1) Design test cases that do not validate the same functionality (prevent redundancy).
2) Select test cases for automation.
3) Select automated test cases for regression testing.
4) Save time by executing only those regression test cases that are affected by a change in the source code.

## II. Providing Insight for the Tester

What we think UI testing lacks is the reliable metric of measurement. The tester cannot be sure how many test cases they need to design and execute to fully cover all of the system. There is also a difference if test cases are designed against the

specification or the implementation. The tester may say, that they designed a test case for every user story specified for the software, but we do not really know how much of the implemented source code is actually tested.

In comparison, for measuring unit testing we use *code coverage* metrics to measure lines, statements, execution branches and functions[1] covered by tests [6]. But aim of the test cases is to test small amount of code called units, usually functions. With code coverage metric the tester knows how much of a function was tested and if another test case is needed. Also the tester can estimate how many test cases are needed to cover all functions – entire implementation. This is not so simple in UI testing.

UI test case is a scenario with steps. Every step is an action performed on the UI of the application. Every action can trigger one or more functions on the UI. Functions on the UI can be, for example, calls to the REST API server. Such call triggers action on the server side, which is another function. The scope of the executed source code is so big, that we can no longer speak about small isolated units. UI test case tests multiple functions on the UI, multiple functions on the server side and also the communication between them. Therefore we cannot easily estimate how many "units" are left untested. But we still can benefit from the code coverage analysis to identify uncovered flows in the source code.

Coverage analysis of UI code has already been used in various UI test automations to enhance efficiency of automation tools. Arnatovich, Ngo, Kuan and Soh [8] proposed an approach to automate UI testing on Android applications to achieve higher code coverage by exercising all of the widgets. Wang, Yang, Xu and Xie in their paper [9] identified exploration tarpits by analyzing UI traces during automation testing. They stated that state-of-the-art mobile UI testing tools produce low code coverage. Zou, Fand, Chen, Zhang and Zhao in their paper [10] proposed a special coverage criteria for web application testing, which combines source code coverage with HTML element coverage, meaning that they not only analysed executed functions, but also how many HTML elements of the whole DOM tree were investigated during testing.

In this paper we focus on manual test designers to help them face challenges that we further explain in following subsections.

### A. Identify Untested Software

If there are lines of code that are not tested by any UI test case, the tester knows that work is not done. It means that there are still some features that were implemented by developers, but were never tested. It can be caused by one of at least three reasons:

- some of the features were never specified, or
- specification changed and some features are no longer needed,
- there are still unidentified test cases for some UI flows.

[1]https://istanbul.js.org/

Some features can be implemented only by assumptions of the developers leaving specification incomplete. For example, if the application handles user login and register flow, there probably should be an option to reset password for users, if they forgot. This feature can be assumed but developers and implemented without the need of specification and detailed acceptance criteria. If test cases are designed against specification, this part of the system may be left untested.

In agile environment the specification can change during development. It may occur that some features are no longer needed and should be deactivated. In this case, developers remove calls of related functions and may not delete the unused source code. The old features are still implemented, but cannot be tested by the tester.

Code coverage analysis on the UI can show uncovered files, lines or functions that were never tested. To identify reasons of uncovered parts we need someone that understands source code to interpret it to the tester. The tester, who is not familiar with the source code, will not benefit just from the information about untested files or lines. Code coverage can help in multiple ways, for developers to find some dead code, and for testers to find either untested software, or missing specification.

### B. Prevent Redundancies

As we mentioned in section I, testing only by specification can introduce significant redundancies into test cases. This is because the tester tries to verify everything what they see on the UI, but some tested parts uses the same implementation. For example, the tester may design test case for every form on the web application, but in the implementation all of the forms use the same component. This way we do not learn anything new from the additional test cases, because component was already tested by another test case.

To prevent redundancies we need to analyse code coverage of every test case separately and compare the results to see the overlap. If many test cases test the same part of implementation, the tester should probably focus on different features of the UI. Saputra and Katayama [11] measured unit test case similarity by their code coverage using machine learning methods. They confirmed reduction of redundant test cases to minimize execution time and enhance quality.

Measurement of similarity between test cases can help the UI tester during design phase. Any new designed test case should be compared with existing test cases and the similarity should show the tester that this test case is redundant with another. This way we can encourage the tester to try different approach to find test case that is more efficient.

### C. Prioritize Test Cases for Automation

Code coverage of UI test cases can be useful to determine what test cases are worth automating first. We can use similar approach as in previous subsection for redundancy prevention. From the group of similar test cases we can choose one with the most covered lines of code.

Another benefit of code coverage is detection of changes. When new change is introduced in the source code, we can find test cases affected by the change based on their code coverage and include test cases into regression only if we benefit from the result.

## III. The Experiment

Usually unit test frameworks contain tool for analysing code coverage. Analysis is run during testing to see how much code is covered by all tests. We intend to analyse connection between the source code and the test case, so we will focus on collecting coverage data for every test case separately. For further research of the challenges stated in previous section we need to collect coverage data from the project that face such challenges. We prepared infrastructure to collect data from code coverage analysis on the UI source code.

For purpose of this research we used the commercial web application project that consists of REST API backend application, web user interface and other modules. The UI is implemented in TypeScript using Angular framework. Size of the UI is more than 107 thousand lines of code in 1642 typescript files, and more than 24 thousand lines of HTML code. UI application is deployed and served from S3 cloud storage. Backend infrastructure consist of many lambda functions deployed on AWS platform. Development team consist of 8 full stack developers, 3 machine learning specialists and 3 testers. One of the testers is also test automation programmer.

### A. Testing flow

Testers design end-to-end (UI) test cases for requested features of the application. They are also entrusted to formulate specification for the features. Test cases are grouped by the user story, which has feature description and criteria. If the specification change, test cases are adjusted accordingly and retested to assure that no new faults were introduced (in test case or implementation). Tests are managed using Zephyr Squad tool[2] for Jira Software[3].

Test case include test scenario formulated in sequence of steps, and preconditions that need to be met before testing. Additionally, it may contain more detailed precondition description for automation, like username and password for the user dedicated to automation test.

These test cases are executed manually and some of them are selected for automation. Automation testing is handled by Robot framework[4] with Selenium Library. Robot framework allows to use simple domain specific language similar to natural language. However, automation tester still needs to locate objects on the web page and define them in automation test cases. Also automation tester needs to handle more sophisticated operations, like click-and-wait action on the UI.

For comparison, test scenario from test case defined in Zephyr tool is showed in table I and automation test case from Robot Framework is showed in table II. We would like to note that steps in Robot Framework test case are custom keywords, those keywords define sub-steps needed to accomplish specific action. Sub-steps can be locating elements, checking values and waiting for elements to be visible.

TABLE I
Test scenario in test case

| - | **Test Step** | **Test Result** |
|---|---|---|
| 1 | Tap on Sign Up button (green). | Signup page is loaded. |
| 2 | Check *Sign up* section. | *Sign up* section contains: ... |
| 3 | Fill email. | |
| 4 | Click on Sign Up button. | ... |

TABLE II
Test case in Robot Framework

| **Test case name** | |
|---|---|
| Click | ${signup_btn} |
| Check Sign up section | |
| Fill email | email@example.com |
| Click | ${signup_btn} |

While test cases from Zephyr tool are executed and evaluated manually by manual testers, Robot Framework test cases are executed automatically when source code is updated. Currently all up-to-date test cases are included in regression testing.

### B. Collecting Code Coverage Data

Our automation test cases are executed on fully deployed application, from which we cannot easily extract coverage information. To collect code coverage data from test executions we prepared an infrastructure, that is used in a software clone. In software clone we adjusted UI of the application so it can provide information about executed lines of the source code.

As we mentioned earlier, application is programmed in TypeScript language. If we want to know which lines of the source code were executed, we need to *instrument* the code with additional functions. For this we used command line tool NYC[5]. Tool works the best on the JavaScript code, so we had to build the application before instrumenting.

Instrumented code contains function before each statement, line or condition. Every function increment one of the following counters:

- lines,
- statements,
- branches, and
- functions.

Built application code by default does not provide connection between statements in the built code and original TypeScript code. In order to actually see how many lines of the original source code is covered, we need to enable source maps to connect transpiled code to original code. Then NYC tool needs to recalculate source maps again to match instrumented code.

[2]https://smartbear.com/test-management/zephyr-squad/
[3]https://www.atlassian.com/software/jira
[4]https://robotframework.org/
[5]https://istanbul.js.org/

NYC tool identifies every instrumented source file with its absolute path. This is an issue if we instrument code on some machine and then work with the coverage data on another. When we downloaded coverage data (which is a JSON object) from the web browser on our local machine, we could not generate a readable coverage report, because reporter tool could not validate the path of the files. However, existence of the actual files was not required to generate the report, only their base paths (project path) was validated. We solved this behavior by adjusting NYC tool to work only with relative paths.

We added additional step to every automated test case. This step downloads coverage data JSON object from the browser when test case is finished. We are interested in code coverage even if the test case fails, so we defined tear-down step, which will be executed on the end of any test case. Coverage data object is saved into JSON file named the same as the test case, so we can pair them when we will analyse them later. After JSON file is created, we generate coverage report from it and include it in the Robot Framework test report. Coverage report is showed in figure 1.

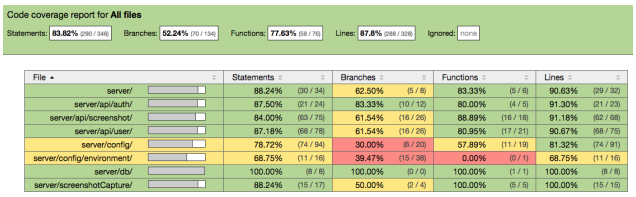


Fig. 1. Example of code coverage report in LCOV HTML format.

Robot Framework report, Code Coverage reports and original code coverage JSON files are stored on S3 cloud storage for further analysis.

### C. Observation

Using our infrastructure we collected code coverage data for further analysis. We observed that percentage of code coverage is very similar for every test case (about 22%), even if they test entirely different features. Most of it is probably caused by same setup step, which is logging into the application. Currently it is necessary to handle login separately for every test case.

To solve this we need to reset coverage counters (see subsection III-B) after test preconditions are met, for example after user is logged into the system. Other solution could be to label code coverage from just login action and subtract it from the final coverage.

## IV. Conclusion

In this paper we described infrastructure for collecting code coverage data from UI test cases designed by manual testers. For the research we used commercial web application project with UI developed in Angular framework. Collection is realized by test automation and resulting code coverage data are saved for further analysis.

We plan to use this data to address challenges mentioned in sections I and II. We believe that we can help testers design more efficient test cases if we provide them information about the impact of their test cases on implementation.

## Acknowledgment

This work was supported by project VEGA No. 1/0630/22 “Lowering Programmers’ Cognitive Load Using Context-Dependent Dialogs”

## References

[1] S. K. Singh, and A. Singh, “Software testing,” Vandana Publications, 2012.
[2] Strategic Planning. “The Economic Impacts of Inadequate Infrastructure for Software Testing,” in National Institute of Standards and Technology, May 2002.
[3] M. Hossain, “Challenges Of Software Quality Assurance And Testing,” in International Journal of Software Engineering and Computer Systems, vol. 4.1, pp. 133–144, 2018.
[4] I. Hooda, and R. S. Chhillar, “Software test process, testing types and techniques,” in International Journal of Computer Applications, vol. 111.13, 2015.
[5] A. Contan, C. Dehelean, and L. Miclea, “Test automation pyramid from theory to practice,” IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 1–5, 2018.
[6] D. Richardson, O. O’Malley, and C. Tittle. “Approaches to specification-based testing,” in Proceedings of the ACM SIGSOFT’89 third symposium on Software testing, analysis, and verification, pp. 86–96, 1989.
[7] G. Grano, C. De laco, F. Palomba, and G. C. Gall, “Pizza versus pinsa: On the perception and measurability of unit test code quality,” International Conference on Software Maintenance and Evolution (ICSME), pp. 336–347, 2020.
[8] Y. L. Arnatovich, M. N. Ngo, T. H. B. Kuan, and C. Soh, “Achieving high code coverage in android ui testing via automated widget exercising,” in 23rd Asia-Pacific Software Engineering Conference (APSEC), IEEE, pp. 193–200, 2016.
[9] W. Wang, W. Yang, T. Xu, and T. Xie, “Vet: identifying and avoiding UI exploration tarpits,” in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 86–94, August 2021.
[10] Y. Zou, C. Fang, Z. Chen, X. Zhang, and Z. Zhao, “A Hybrid Coverage Criterion for DynamicWeb Testing,” in SEKE, pp. 210–213, 2013.
[11] M. C. Saputra and T. Katayama,“Code Coverage Similarity Measurement Using Machine Learning for Test Cases Minimization,” IEEE 9th Global Conference on Consumer Electronics (GCCE), pp. 287-291, 2020