POLITECNICO DI TORINO

Master degree course in Software Engineering

Master Degree Thesis

Web Testing Fragility

A Study on Open-Source Web Applications

Supervisors
prof. Luca Ardito
dipl. ing. Riccardo Coppola

Candidate
Mehrnoosh Rigi
registration: S236436

ACADEMIC YEAR 2019-2020

This work is subject to the Creative Commons Licence

Summary

Web Testing Fragility

A Study on Open-Source Web Applications

Due to the vast majority of web applications, the quality of produced code is very important to make sure the system works as expected. Web applications consist of a client-server model. Which client in this case is a browser that asks for the services from the server through the API. correct functionality of the system can be guaranteed as testing. Testing can be done manually or automatically. Manual tests are done by a quality insurance team or a person. It is based on box approaches. Namely: black box testing, white box testing, and gray box testing. Black box testing focus on the external expectation of the behavior of the system and the internal working of the application is not required to be known, the tester views the program as a black box and is just concerned about the input and output of the program while the white box tests require full knowledge of the internal working of the application and focus in the code, the internal structure of the program the logic and the inner working. It's suitable for algorithm testing. Instead, gray box testing is a mixed feature of black box and white box testing. The tester can leverage the application's internal working, studies the requirements specifications, and communicates with the developer to understand the system's internal structure.

Automatically testing is faster with fewer errors of human in comparison with manual testing. It is faster and cheaper. makes assurance of different levels from unit test to end to end tests of the quality of software. We can consider the level of testing as a pyramid which in this case the speed of the test is very important. The vast majority and base of the pyramid should be dedicated to unit tests or functional tests because it is the fastest way of testing. The best case for writing a unit test is TDD, which is an acronym for Test Development Driven and means write the test first then produce the code. it's famous as well with red, green, and red. Red means the test fails

because the code is still not delivered, Green means the test passed after the development of the code and red again means each update in the code should be at the update on the test.
Another testing level is end-to-end tests which take more time but widely help for regression tests and nightly running tests. End-to-end tests sometimes contain also integration tests. Because it tests user interaction through the browser, which can also make API requests. but from another side integration tests are more important for working along services inside the web application. There are many other testing methodologies like stress tests, security tests, database tests and etc. but the focus of this study is just based on scripted tests.
The development of the project can be done by many methodologies such as waterfall, agile, and so on. But in this study just coding with help of scripted testing tools is considered despite the development methodology. Developing software with high quality based on the new changes needs high effort for the tester or the developer to update each test plan and redirect to a high-quality direction so it increases the cost of development. Furthermore, maybe it's just not related to updating the old test, maybe it turns to innovate some ideas to add a new test to improve the quality. Even if we consider software testing as a continuous improvement process, in parallel with the development, verifying the consistency between the test and developed code is necessary. In automated testing, especially in scripted type, the challenge of testing refers to three areas:
Creating the test, Processing, And Quality aspect of the test case specification.[11] which any of them is impacted by updated code. Setting up automatically running the test in a DevOps environment is very important for regression tests.
the two most important tools related to the Application Under Test(AUT) are considered in this study. Namely selenium, Cypress. The aim of this study is fragility measurement testing scripted tools among the developed code. The term Fragility refers to the tests that fail or they need maintenance due to the development process. Refactoring the tests shows that the tests are also updated based on any update inside the code. Adding new features needs effort for adding new tests or change requests (CR) of a feature that needs any update on the written tests. The modification of tests due to the life cycle of coding is very important. With growing the tests managing and modifying tests gets difficult. So maintenance of the test especially for huge projects increases the cost. This study is continuous work of the study of Mobile GUI Testing Fragility: A Study on Open-Source Android

Applications[4]. the purpose of the mentioned study was to estimate the influence of changes in the code and besides inducing the changes in the test code and quantifying the fraction of them based on the used metrics in the study.
The first research question is:
How many open-source web applications projects, use the automation frameworks, and how much of the test code coverage for the new feature of the project?
Basically in this study based on a query of the API of the Github repositories, web application repositories that contain at least two tag releases and used the considered testing framework was used. The query was taken with aim of the javascript and saved in a JSON file as a result file. The open-source applications from the year 2016 and later were studied.
Four metrics are calculated for this question:
TD: Tool Diffusion: The adoption size of the tool in the projects.
NTR: Number of tag releases: With help of python script and usage of command the number of tag releases for each repository was taken.
TTL: the python code also takes the Total Tool LOC(Lines Of Code) for each repository in the latest tag release with help of the CLOC command.
TLR: Test Lines of code Ratio was calculated for each test file and file which imported those test files and the summation of them in every tag release.
The second research question is:

How much the test codes are getting aligned with the new update and new modifications in releases of the project?
To respond to this question four metrics have been calculated:
MTLR: Modified Test Lines of code Ratio. Which is the division of tag differences of the test file with the previous tag and the total number of lines of code with the previous tag. This metric lies between 0 and 1. Which 1 means the number of updates on tests was significantly high.
MRTL: Modified Relative Tool LOC. this metric was computed for each tag release the division of tag differences of each test file by the total production LOC.
TMR: Tool Modification Relative Ratio. Is for the current tag the division of MRTL by the TLR previous tag.
MMR: Modified Release Ratio. computed as the ratio between the number of tagged releases in which at least a file associated with a specific testing tool has been modified, and the total amount of tagged releases featuring the file associated with that tool.

All metrics were computed by the python implementation and saved in a JSON file to extract the final result and graphs. So far the huge amount of 0 number for the calculation means the fragility of test files is higher than expected.

Acknowledgements

I candidati ringraziano vivamente il Granduca di Toscana per i mezzi messi loro a disposizione, ed il signor Von Braun, assistente del prof. Albert Einstein, per le informazioni riservate che egli ha gentilmente fornito loro, e per le utili discussioni che hanno permesso ai candidati di evitare di riscoprire l'acqua calda.

Contents

List of Tables

List of Figures

Part I

First Part

Chapter 1

Introduction

1.1 Testing web applications

With intensive usage of online platforms and their significant impact caused accomplished reliable and secure applications which will be working as expected, consequently, many works are substituted by human beings. However, for controlling the quality, the application must be tested and evaluated to detect the defects and then correct them. [1]

A web application is consist of a client which takes some services from servers through the API. API stands for Application Program Interface, which acts as a middle ware between components of software and provides protocols and tools to allow components to interact between themselves. The quality of correct working components should be guaranteed in order to make sure that the system is working as expected.

Testing the system and software as a web could be done at any level, from unit testing till the end to end testing. The levels of testing are like a pyramid. Let's say the unit testing should have a bigger surface in comparison with other levels. because it's faster and also tests the granularity and functionality of services. which will be explained in detail.

For the sake of delivering high-quality tools and declining costs, automation tests would be required. Also, because software, usually may be carried out far from the customer's needs, it will guarantee the regression of test cases. However, it's essential in order to ensure the high quality of software, the test should compose well written to be error-prone and find the errors and bugs. Generally, Software testing consumes 30 to 60 percent of all life cycle costs, depending on the product's criticality and complexity.[2]

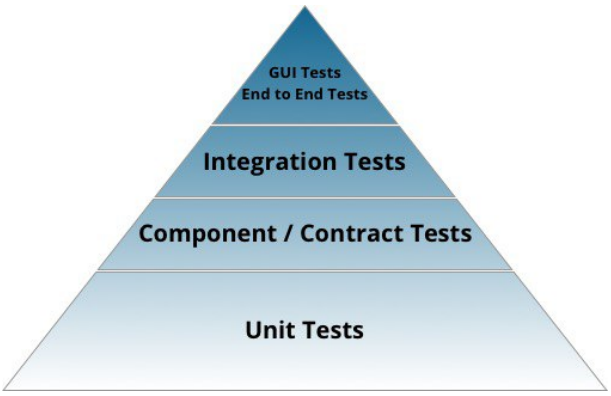

Figure 1.1. Pyramid of testing levels

More attention should be paid to web application testing efficiency, usability, and comprehensiveness.[3] Accordingly, the fragility of written test cases is also vital. Based on that the amount of updating the test codes alongside the software development life cycle, tracking the modifications that were needed by test classes to be executable on different releases. [4] is also important. The coordination of test codes with the software development is considered to compute the fragility of test scripts with the developed application.

Chapter 2

Review and Background

2.1 Web Applications

The Internet is no longer just a static page, it requires active user interaction to improve the functionality of the requested pages on the internet. We can say that a web application is just like a normal application but on the internet and also without the cost of installation. we use it at home for private reasons, and we use it at work for professional reasons; we use it for fun (e.g., gaming) and for serious interactions (e.g., home banking), via fixed stations and mobile devices and these are just a few of the motivations for and the contexts in which we exploit such a powerful medium. The first Web site, created by Tim Berners-Lee and Robert Cailliau at CERN (European Nuclear Research Center), consisted of a collection of documents with static content, encoded in the HyperText Markup Language (HTML).[**?**]

Traditionally web applications are divided into two parts: The server side and client side, have their own programming languages that interconnect themselves to transfer data and requests. It means that the client sends requests through methods that based on the request are common:

- GET: The request sends in order to get some information about the specific URI(Uniform Resource Locator) to retrieve or view the file or program or even a simple HTTP page.
- POST: The POST method contains the body, which sends data from the body to the server, for example, login user, sending the encrypted credential through the safe channel to the server.
- PUT: When the method of PUT is used, it means the client already

knows some necessary information. The mentioned information depends on the necessary data for the server, like the token of the user that already is authenticated. For example, the request is updating the authenticated user profile, the updated information sends through the body to the server with the token generated already for the user after login. it means updating the database for this user.

- DELETE: It's a request to remove data on the URI.

Or even a noun request like Head or Option. The requests are sent through the header followed by the body. The header is divided into several groups:

- General Headers: There is no relation between request and response and it applies to both request and responses.
- Request Headers: It has more information about the resource.
- Response Headers: It has additional information about the response which sends.
- Entity Headers: These contain information about the body of the resource.

The Body, Is followed by the header when the request needs to send data, but the request that fetches data or resources doesn't need to have the body. The body can contain one part or multiple parts.

The response to the request also contains a header and body. Also, it contains the status of the request, The status code means the situation of response, For example, 200 is OK, 400 means Bad requests, and so on. The header and body are like the sending request.

But it's necessary to mention that, with an improvement in technology and some problems in security like code injection, multiple server-side and client-side dynamic contents are significantly get considered.

With increasingly massive and broad applications of Cloud-based development, many diversified APIs [1] are emerging. Web APIs for Internet software, provide a natural way to wrap and deliver software functions as self-contained services that can be accessed through standard protocols. By serving as

[1] Application Programming Interface

the contracts between service providers and service users, APIs can effectively shield heterogeneity as well as enforce decoupling. However, the inherent open, collaborative, and dynamic characteristics of Web APIs raise new threats to the quality of systems developed by composing services. API testing is thus gaining more and more attention. Due to their wide impact, any flaw in the cloud APIs may lead to serious consequences. API testing is thus necessary to ensure the availability, reliability, and stability of cloud services. [5].

2.2 Testing web applications

Software quality assurance: The degree to which a software product meets established requirements and includes software testing, quality control, and software configuration management; however, quality depends upon the degree to which established requirements accurately represent stakeholder needs, wants, and expectations.

Software quality control:

- A set of activities designed to evaluate the quality of a developed or manufactured product. Contrast with software quality assurance.
- The process of verifying one's own work or that of coworker. [6]

Testing the software includes many levels. namely:

- Unit testing: Unit testing refers to small testing which is done mostly by developers to guarantee the functionality of written sections. for example, testing the function returns as expected with each type of input.
- Integration testing: In comparison with unit testing, integration testing is in larger aggregate. it tests the integrity of all services that work together correctly.
- End to End testing: This type of testing is purposed to check from starting until the end, data and information are passed between various system components and services as expected.
- System testing: This level of testing assured that the completed system meets the specified requirements.

- Acceptance testing: Also known as "definition of done" refers to testing the user story based on the predefined requirements of the task or feature of the done story.

And the testing methods of software includes:

- Static testing: Refers to executing before compilation (it's time-independent), it's less expensive and it's for preventing defects, however, it requires loads of meetings.
- Dynamic testing: Is done by executing the program and after the compilation, it tries to find and fix the defects it's more expensive, but it requires fewer meetings (time-dependent).
- Box approach:

    1. Black box testing: In black box testing the internal working of the application is not required to be known. Testing is based on external expectations. It's based on the bases of programs and system functionality, the tester just observes the input and output of the program. The tester focuses on testing the functionality of the program against the specification. Black box testing focuses on testing the program's functionality against the specification. With black-box testing, the tester views the program as a black box and is completely unconcerned with the internal structure of the program or system. Some examples in this category include decision tables, equivalence partitioning, range testing, boundary value testing, database integrity testing, cause-effect graphing, orthogonal array testing, array, and table testing, exception testing, limit testing, and random testing.[7]. Black box testing is not appropriate for algorithm testing, and the test is based on trial and error methods and is testing what the program or software suppose to do.
    2. White box testing: It requires full knowledge of the internal working of the application, and its structural testing and it focuses on produced code, so the data domains is better tested. It's appropriate for algorithm testing. The tester examines the internal structure of the program or system. Test data is driven by examining the logic of the program or system, without concern for the program or system requirements. The tester knows the internal program structure and logic, just as a car mechanic knows the inner workings of an

automobile. Specific examples in this category include basis path analysis, statement coverage, branch coverage, condition coverage, and branch/condition coverage. [7].

3. Grey box testing: It contains mixed features of black box testing and white box testing (functional and structural testing). The tester has to limit knowledge of the application's internal workings, study the requirements specifications, and communicate with the developer to understand the system's internal structure. The motivation is to clear up ambiguous specifications and "read between the lines" to design implied tests. [7]. The testing is at the base of a high-level data flow diagram.

- Manual and Automated Testing: Manual testing is done without the help of tools while automation testing is written as a scripted code. Manual testing is less accurate because of human being errors while it can fetch hidden information. On contrary the automatic testing can do regression tests and it's faster and cheaper and decline the costs. However, they are many possible classifications of software testing which are not just divided into mentioned methods. [7]. The taxonomy of software testing is not summarised just in the mentioned methodology. they are extended more than 40 techniques based on the requirements those techniques are used in software development testing quality, like security testing, acceptance testing, stress testing, database testing and etc. Based on the subject, we mention here the automated scripted testing of tests.

2.3 Software Development Methodology And Testing

Also known as Software Development Life Cycle (SDLC), is the process of dividing the development of software into distinct phases to improve the project and product management and also the design and development of the software. The most important and used developing methodologies are divided as follows:

1. Waterfall Methodology: The traditional method of development was the waterfall method. in the waterfall method, the lifestyle of development is divided into units and each phase started after the previous phase is finished. in the waterfall method the phase of Plan, Do, and Check, could

also be applied in the testing and quality and it's typically considered as a separate approach. Theoretically during the development, after one phase is finished, going back and checking the previous phases should not have happened. The phases are containing:

(a) User requirements: In this phase, the requirements are checked and analyzed by the user and as a consequence, for the next step a document is prepared by the user.

(b) Logical design: In this phase *Entity relation diagrams*, *Process decomposition diagrams and* data follow diagrams are designed to provide better view of data and the functional progress for managing them.

(c) Physical design: The output of logical design is used in this phase to provide the design of physical units like database schema.

(d) Program unit design: According to the output of physical design, the programmers develop the system and structure to enter the coding phase.

(e) Coding: In this phase, the coding, writing unit testing, integration testing, end to end testing is done.

(f) Operation and maintenance: The product is delivered to the user and the acceptance testing is done, also maybe some new features adding or even some change requests will come to maintain the software.

In the waterfall methodology is the whole effort to completely separate the testing process and development process. that's because of the separated phases in a waterfall, the testing methodology uses the designed documentation of the first phase(User requirements) to design and plan the test cases. as a consequence, the programmer does not test their own code by their view and it's tested by the tester. because finding the defects from the programmer's point of view is very difficult. Hence we consider the testing and application development as concurrent working and we expect the testing phase starts early, in the user requirements phase, the tester verifies acceptance testing, in the logical design, verifies system testing, in the physical design, verifies integration testing, in the program unit design the unit testing is verified. the mentioned phases by themselves are going as a continuous improvement process. for each phase the steps contain:

(a) Planning
(b) Overview
(c) Preparation
(d) Inspection
(e) Rework
(f) Follow-up [7]

A problem that might have an effect on testing is that sometimes some phases of waterfall methodology are skipped for example designing the requirements, then immediately the phase of coding is started. This might increase the risk of changing code meanwhile development and then the test also should be changed in order to be compatible.

2. Spiral Methodology:

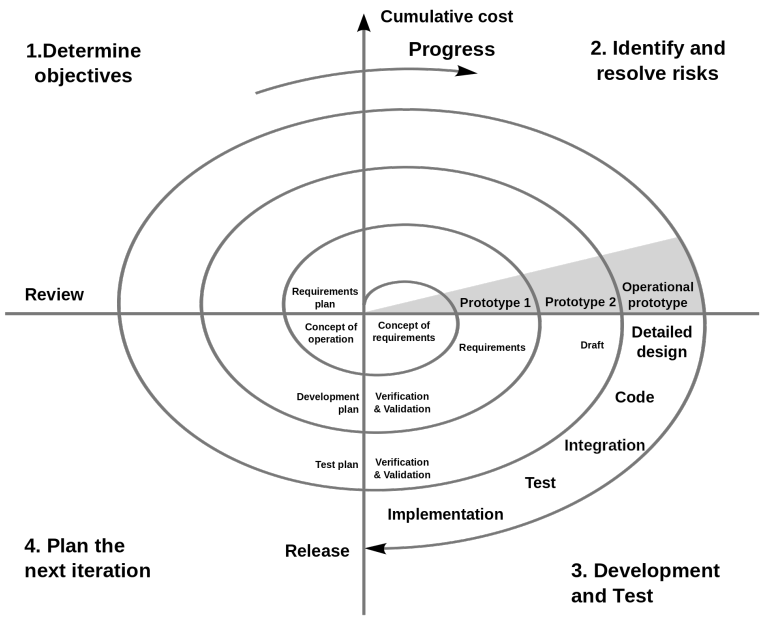

Figure 2.1. Spiral model, Boehm 1988

[13]

In the spiral methodology, products, as it's shown, it's like waterfall methodology that develops in sequential phases. The common problem of waterfall methodology is that the elapsed time for the development of software, can be affected by the user at the very beginning or very

end of development, and consequently, the delivered developed software is not exactly same as the what is requested. In spiral methodology, a small part and functional part of the requested system is developed and delivered, and it's iterated over the all development process. as a result, the user receives a part of the software more in advance and can send feedback to the developers and developing and testing are going dynamically. As can be seen from figure 2.1, cycles or spirals are in period of time and phases inside cycles are repeated subsequently.

Testing in the spiral methodology requires that the tester enter inside the spiral and must be very close to the programmers. another necessary test due to the iteration inside the spiral is the automated regression test, which needs sufficient time and resources by itself. Spiral testing is initiated from the base and growing incrementally concurrently in the developing software. as soon as the test will be completed and the functionality of the application will be verified, testing of all systems and acceptance criteria will start. [7]

3. Prototyping: As it's obvious by its name, in this methodology, a prototype version like a trial version of the software is developed and delivered, but there isn't any idea about the developed software in a real domain. this version allows making decisions for stakeholders to continue and complete the final version.[7] The testing phase in a prototype is after building the software model and before needing the reworking the acceptable prototype.

4. Agile: Managing large projects in waterfall methodology is very difficult because it rarely happens that each phase is isolated and closed before other phases. in reality there are always circular loops to back to previous or earlier projects to adjust and maintain the phase. In large distributing teams there is a problem with communication and centralizing the management of the project and as a consequence reaching the final test phase. The tester finds more bugs and the cost of fixing them increases due to the capacity of developers for handling the coding and meanwhile fixing bugs. With mentioned difficulty, a new methodology appeared as an agile name, but it's important to mention that agile methodology is not just a single or group of rules that should be obeyed in the life cycle of development, instead, it makes up lightweight management, engineering development techniques by the on hand tools. the agile process doesn't use a dedicated way for the process of development,

instead, it shapes by the team and improving constantly by the developed code. growing the development process may cause complexity, so simplicity is necessary to make the process flexible. In agile we have a storage full of thoughts of team members so there are many practices that can be changed in every process, which means there isn't any fixed practice that we use for all processes. As we mentioned in agile the management is lightweight, which means instead of managing in a top-down approach, the management goes in reverse a bottom-up. It means the organization is team-based that cause filling the gaps between the processes and reflects the functionality to the team members.[8]

5. Scrum: Scrum is introduced to simplify the project manager and it's a proposal for repetitive development. scrum consists of three roles which are:

   (a) Team members: Includes the developers and testers.

   (b) Scrum master: Responsible for organization of the process of development and the efficiency of team members that verifies problems get resolved and the job is done in high quality.

   (c) Product owner: includes stakeholders. The product owner is responsible to represent and clarify the requested result of stakeholders.

   There is a backlog in the process of development that contains all requirements and their priority. The level of development is divided into timetables which are known as *Sprint. In sprint tasks and user, stories are defined, which at the end of the sprint they should be tested and delivered. Each sprint consists of a couple of meeting like* Sprint planning, in which in this meeting the story point and timing for the development and testing are defined. Every day there is *Daily scrum which a brief of work that is done is exchanged between teams. Then at the end of the sprint, there is* Retrospective meeting in order to have feedback on the sprint between the team and scrum master for the purpose of increasing and improving the efficiency.

2.4 Challenge for testing web applications

Testing by itself is a difficult effort because it needs to find and defect as much as possible all problems and errors of the developed parts. The tester needs to think from another point of view(as a real user or even as a 3rd

party), in order to find out all possibilities and detect the defects of all paths and directions of the software.

In the life cycle of development especially in the agile method, the feature or task of development becomes a change request by the stakeholders and this caused changing the feature and dealing with short delivery times. As a consequence, systems are delivered without being tested or without updating the old tests, because developing the software with quality based on the new changes needs high effort for the tester or the developer to update each test plan and redirect to a high-quality direction so it increases the cost of development. Furthermore, maybe it's just not related to updating the old test, maybe it turns to innovate some ideas to add a new test to improve the quality. Even if we consider software testing as a continuous improvement process, in parallel with development, verifying the consistency between the test and developed code is necessary. Hence the test should be simple, complete, and fully functional coverage, without bugs on one hand and in another hand be compatible and updated to the developed code. Sometimes the expectation of customers is very vague, even the customers don't know what is do they need. this causes difficulty in testing as a high quality and as an expectation and understanding the business environment. In the process of testing, with time slots of developing, testing, and delivering, management of skipping or not getting into specifics of some task, on its own is another challenge of testing. [**?**]

Exclusively, in web applications, usually with the existence of third parties, the structure of the domain, different execution platforms, and so on, extra knowledge is required. especially for the manual tester, having technical knowledge in order to have a better understanding and consequently keep updating the tests with new development is vital to achieving the wide coverage of testing.

In automated testing, especially in scripted type, the challenge of testing refers to three areas: *Creating* the test, *Processing* and *Quality* aspect of test case specification.[10] which any of them is impacted by updated code.

When the automation test is running inside the DevOps environment, Setting up automatically the automation test environment is vital for effective consequences. Resetting means that goes to the initial state of the test otherwise it could influence to the result of another test when the execution is in order.[11] In testing GUI[2] web applications, *Capture and Relay*, is one

[2]Graphical User Interface

trend that the tester provides a beset assumption and path and record to make the script for automated testing. The flow path contains navigation, mouse click, button pressing, or in general all events of the user interaction for delivering a script of test cases to automated test. CR support automatic regression testing. *Programming web testing*, As it's mentioned, it's based on script testing of manual testing. It contains test cases that are written by developers and can be automated. Maintaining the programming is much more expensive but it's easier in comparison with CR.[12]

2.5 Fragility measurement

The terms of *Fragility* refers to the tests that fail or they need maintenance due to the development process. Hence refactoring the scripted test is necessary to make the test adoptive with maintenance. it's supposed that the test is modified due to modification of code, or the test is added due to the new functionality which is added to the code.[4] Adding the new tests cause growing the tests and as a consequence managing and modifying gets difficult.

The CR[3] trend in web application testing, is much more fragile in comparison with programming. because it's based on the record, a minor change in the codes tends to break previously recorded testing. When the code has been updated, the test case may be broken. it needs modifying. Especially when it's related to functionality the effort of updating is much more than structural or layout testing. [12]

[3]Capture and Relay

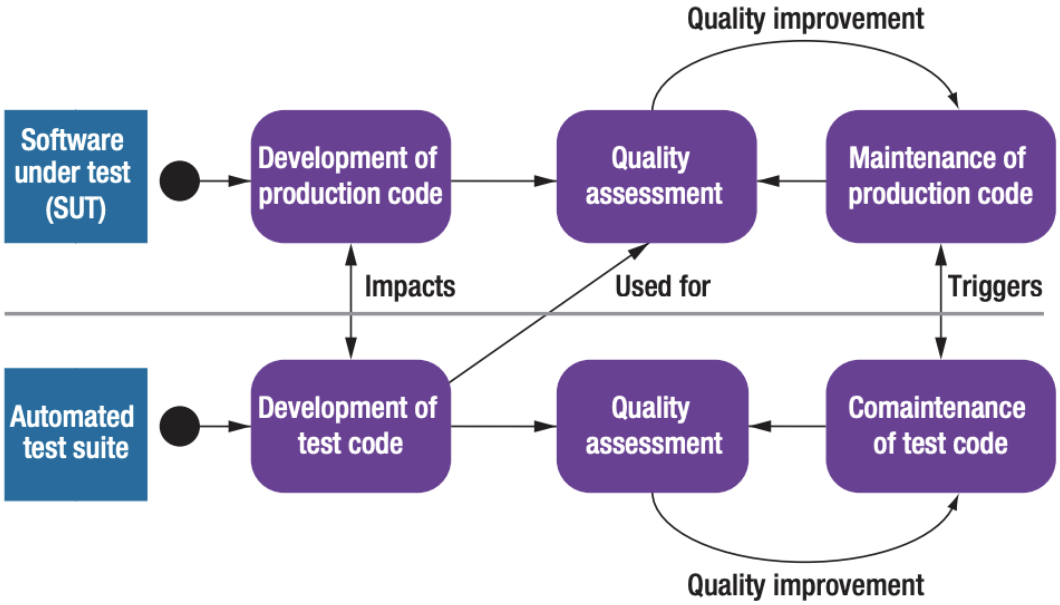

Figure 2.2. An overview of software test code engineering (STCE)

Software Test Code Engineering [14]

When a test needs to be updated, to prevent the fragility of the test, the test case needs really analyzed for checking if it really detected the defects or if it's just a false positive. maintenance of the tests especially for huge projects with continuous delivery increase the cost.[14]

From another perspective, in the continuous delivery, when the environments[4] are different in developing process and writing new tests of the developed code, increasing the complexity in the management of test cases, furthermore when those have their own languages, it causes the cost of modifying and maintenance double or triple. So as a consequence, the fragility of tests also in all environments and possible languages for the project, is something that happens regularly. In this case, engineering the test case in order to make them simple and efficient, with high possible coverage, meanwhile of the development process, is very vital for the maintenance and life cycle of code development and test codes.

Also could be effective to mention, besides the fragility of tests, to maintain

[4]environments refer to development, production, stage and ...

the tests, for better management, writing beautifully has many effects in time of updating the code. For example, when iterated works of testing define a function, and use it in all parts, changing that function is easier in comparison with changing all parts of the code.

Chapter 3

Methods and Study Design

This study is designed in order to continue the work of *Mobile GUI Testing Fragility: A Study on Open-Source Android Applications*[4]. The purpose of the mentioned study was to estimate the influence of changes in the code and induce the changes in the test code and quantify the fraction of them based on user metrics in the study. but the current research is done in Web application software. Same as working of *Mobile GUI Testing Fragility: A Study on Open-Source Android Applications*[4], in accordance with the mentioned article and its acquirement the following questions will be answered by this research:

1. **RQ1 Diffusion:** How many open-source web applications projects, use the automation scripted frameworks, and how much of the test code has coverage for the new feature of the project?

2. **RQ2 Evolution:** How much the test codes are getting aligned with the new update and new modifications in releases of the project?

The first step is the estimation of the diffusion of web application scripted automation frameworks. The first step was taking the open source real web application projects from the GitHub repositories, which is done by taking the query from the mentioned API and searching the repositories that they used our considered testing frameworks in order to sanitize the search repositories. Then studied the number of changes of the application codes and test codes through the releases history. the study was done by the comparison of the file by file in the repository. In the end, with the aid of Python, I tracked

the modification of each test file and test methods to compute the set of changes which it helps with the computation of metrics that we considered as an indicator for the research.

Finally, the test codes that have been modified, are extracted, in order to be manually got examined to construct a taxonomy of modification reasons and compute the frequency of occurrence of individual causes.

3.1 Metric definition

As the metrics were defined in the previous work in *Mobile GUI Testing Fragility: A Study on Open-Source Android Applications*[4], I reuse those metrics again for the study of the fragility of tests methods and test cases in the web application open-source projects. Those metrics are divided into the following groups:

1. **Diffusion and Size Metrics:** The Diffusion defines the amount of test code while the Metrics define the size of the tested project.

2. **Test Evolution Metrics:** It defines the evolution of test code during the process of growing the web application project.

The following table depicts the metrics and a short description of the acronyms and immediately after the table, the metrics are explained in detail.

Group	Name	Explanation
Diffusion And Size (RQ1)	TD	Tool Diffusion
	NTR	Number of Tagged Releases
	TTL	Total Test LOCs
	TLR	Test LOCs Ratio
Test Evolution (RQ2)	MTLR	Modified Test LOCs Ratio
	MRTL	Modified Relative Test LOCs
	TMR	Test Modification Relevance Ratio
	MRR	Modified Release Ratio

Table 3.1. Metrics Definition

[4]

3.1.1 Diffusion and Size(RQ1)

For defining the amount of test code and the size of the tested project, the following metrics were used:

1. **TD** (Tool Diffusion): The percentage of Web application projects that use the considered testing tool.

2. **NTR** (Number of Tagged Releases): is the number of tagged releases of each Web project (i.e., the ones that are listed by using the command git tag on the GIT repository). This metric can be used to understand what is the nature of the applications that are more likely tested using GUI Automation Frameworks.

3. **TTL** (Total Test LOCs) is the total number of lines of code that can be attributed to a specific scripted automation framework in a release of a web application project.

4. **TLR** (Test LOCs Ratio) defined as:

$$TLR_i = TTL_i / Plocs_i$$

,

The *Plocsi* is the total amount of **Project LOCs**, LOC[1], contains the project code and also the test codes. This metric is in the range between 0 and 1. [0,1]. This interval defines, the amount of testing code to a considered scripted automation framework.

3.1.2 Test Suite Evolution(RQ2)

Test suit evolution moves forward to compute the development of test suite codes with the development of web applications projects. This computation is on each pair of consecutive tag releases.

1. **MTLR** (Modified Tool LOCs Ratio) defined as:

$$MTLR_i = Tdiff_i / TLOC_{i-1}$$

where *Tdiff* is the amount of added, deleted or modified **LOCs** that can be associated with a specific tool, between tagged releases i - 1

[1]Line Of Codes

and i. This quantifies the number of changes performed on existing LOCs that can be associated with a given tool, for a specific release of a project. A value higher than 1 of the metric means that more lines are added, modified, or removed in test files in the transition between two consecutive tagged releases, than the number of lines already featured by them.

2. **MRTL** (Modified Relative Tool LOCs) defined as:

$$MRTL_i = Tdiff_i / Pdiff_i$$

where *Tdiffi* and *Pdiffi* are respectively the amount of added, deleted, or modified tool and production LOCs, in the transition between release i - 1 and i. It is computed only for releases featuring code associated with a given testing tool

$$(i.e., TRL_i > 0).$$

This metric lies in the [0, 1] range, and values close to 1 imply that a significant portion of the total code churn during the evolution of the application is needed to keep the test files written with a specific tool up to date.

3. **TMR** (Tool Modification Relevance Ratio) defined as:

$$TMR_i = MRTL_i / TLR_{i-1}$$

This ratio can be used as an indicator of the portion of code churn needed to adapt files relative to a given testing tool during the evolution of the application. It is computed only when

$$TLR_{i-1} > 0$$

We consider a value greater than 1 of this metric as an index of greater effort needed in modifying the test code than the actual relevance of testing code, with respect to the modification of application code. On the other hand, we consider lower values of this indicator as evidence of easier adaptability of code associated with a given testing tool to changes in the AUT.

4. **MRR** (Modified Releases Ratio)
   computed as the ratio between the number of tagged releases in which at least a file associated with a specific testing tool has been modified, and the total amount of tagged releases featuring file associated with that tool. This metric lies in the range [0,1] and bigger values indicate minor adaptability of the test files (i.e., the set of test classes associated with a given testing tool) to changes in the AUT.

[4]

3.2 Selected Testing Tools

Since our objective was to document the evolution of testing code of web open-source applications, we focused our study on two Automation frameworks, that allow writing any tests through hand-written code. Automation frameworks typically identify the elements of the web through their properties or the definition of the screens of the app (e.g., the Layout files) and offer to the developers a set of functions that allow performing actions on any components, in addition to assertion statements to verify the current state of the app. I selected two automation frameworks that have been cited in the available literature. A selection criterion for the tools was the ability to produce test scripts in web applications since our metrics considered code comparison with the production code of apps. Since the adaptation of Selenium is a very widely used testing framework for web applications, The most important tool for us is Selenium then Cypress. The two selected tools cover together the principal peculiarities that can be attributed to scripted Automation frameworks, hence the results and discussion about each of them can be representative of other Automation Frameworks with similar characteristics.
Table **3.2** summarizes the features provided by the selected tools. These characteristics partly reflect the ones listed by [15], in their description of scripted testing frameworks, to which we added the support to image recognition. In the table, the columns are dedicated respectively to: the nature (black box, or white box) of test scripts produced; the ability to exercise web apps; the possibility of writing test cases spanning multiple metrics, however, we are still unable to discriminate what is the reason behind the modifications to be performed on the test file. The higher MRTL values for the sets of projects featuring Cypress can be justified by the small size of the sets, and by the nature of the projects examined. While the projects featuring code associated with Selenium and Cypress exhibit close average MTLR values, the average MTLR on the set of projects featuring Cypress is way bigger than Selenium. In general, a higher MRTL should mean minor adaptability of a testing tool to modifications performed on the production code, with more changes needed by the test code as a consequence of changes in the production code. [4]
article

- **Selenium:** provides extensions to emulate user interaction with browsers.

Table 3.2. Characteristics of the selected Testing Frameworks

Characteristics of Selected Testing Framework						
Framework	Black Box	Multi App	C and R	Multi-OS	Level	I
Selenium	Yes	Yes	-	Yes	Unit-level, end-to-end level	
Cypress	Yes	Yes	-	Yes	Unit-level, end-to-end level	

[4]

lets to write interchangeable code for all major web browsers. Selenium brings together browser vendors, engineers, and enthusiasts to further an open discussion around the automation of the web platform. At the core of Selenium is WebDriver, an interface to write instruction sets that can be run interchangeably in many browsers. Once everything is installed everything, only a few lines of code get you inside a browser. WebDriver uses browser automation APIs provided by browser vendors to control the browser and run tests. This is as if a real user is operating the browser. Since WebDriver does not require its API to be compiled with application code; It is not intrusive. Hence, you are testing the same application which you push live.

- **Cypress:** Cypress is a next-generation front-end testing tool built for the modern web. We address the key pain points developers and QA engineers face when testing modern applications. Cypress is most often compared to Selenium; however, Cypress is both fundamentally and architecturally different. Cypress is not constrained by the same restrictions as Selenium. This enables you to write faster, easier, and more reliable tests. Cypress helps to write end-to-end, component, integration, and unit tests. Cypress can test anything that runs in a browser.

3.2.1 Instrument

This section describes the tools and scripts that have been used to extract the set of projects on which we conducted our study, and to collect statistics about them.

1. Data extraction from Github: The procedure for searching on the API Github was with the help of the JavaScript language based on the queries. Since the API query result of Github is limited, searching was based on the dates and ranges then the result is paginated and sorted

sequentially. so inside of the query, repositories that were created from January 2016 until September 2020 with the keyword "Web" in the repositories were extracted and saved based on the dates as a JSON file. the number of results was **142830** repositories for all repositories with the keyword **Web** in their description or readme files.

2. Git Code Search: The GitHub Code Search API allows searching for particular keywords inside a given project. The search can be parameterized using the "filename" parameter, which constrains the search only in files named as indicated (if the parameter is omitted, the keyword is searched in all the files of the repository). The "filename" parameter can also be leveraged to search for the presence of files named in a specific way inside a repository, regardless of the code they contain. The "repo" parameter is used to specify the repository in which the search has to be performed.
   Some limitations apply to the GitHub Code Search API, as explained in the Git Documentation: (i) only the default branch (in most cases the master branch) is considered for the code search; thus, if tests are present in older releases but are removed in the master branch, the project will not be extracted; (ii) only files smaller than 384kb are searchable; (iii) only repositories with fewer than 500,000 files are searchable. The second and third issues may be not very relevant in our context since the size of projects and files considered is typically not so big (with the exception of projects containing whole firm wares).

3. Count of lines of code: We used the open-source cloc tool to count the total lines of code inside a repository (or, in general, a set of files). To compute the number of modifications performed to files of a GitHub project, the git diff command is used, to obtain all the modified, added, and removed lines of code between the two releases considered. By default, the diff command shows the modifications performed to the whole repository; as an alternative, it is possible to specify the full paths of a file for both releases, to obtain the modifications that were performed only on it. The git diff command takes into account also blank lines and rows of comments inside files. [4]

3.3 Applied Procedure

In the following paragraphs, the procedure of research is explained in detail.

3.3.1 Context Definition and Test Code Search (RQ1)

The first operation performed to conduct our study was a definition of our context, i.e. the set of projects that were used for the subsequent investigations. We performed three different steps to extract the set of projects used as our context, the first one being a search for the word “Web Application” or "Web" in descriptions, readme files hosted on GitHub. GitHub Search API has been leveraged for this purpose, using the following search string with aid of Javascript programming:

```
const fetch = require('node-fetch');
const fs = require('fs');

const requestOptions = {
  method: 'GET',
  headers: {
   "Authorization": "Basic ****"
  },
  redirect: 'follow'
};

let finalJson = [];

const callApi = async () => {
 for (let i = 1; i <= 30; i++) {
 const res = await fetch('https://api.github.com/search/repos
 const json = await res.json();
 if (json.items) {
 finalJson = [ ...finalJson, ...json.items ];
 }
 }
}

callApi().then(() => {
 fs.writeFile('September-first-web-apps.json', JSON.stringify
   if (err) return console.log('Failed to save file');
   console.log('File saved');
 });
});
```

This way, we gathered a total of 142,830 GitHub repositories. We then applied a filter to cut out from the context all the projects that have no tagged releases. This is done because the aim of the study is to track the evolution of the considered projects, and – as it is detailed later – differences between tagged releases are computed. Considering that, projects without at least one tagged release (which allows for a single comparison, made between itself and the master release) are not of interest. To find how many tagged releases are featured by a project, the git tag command is used. This way, we obtained a set of 139 repositories for selenium and 59 repositories for cypress projects with a history of tagged releases.
Looking only for the keyword "web" would have included in the results also libraries, utilities, and applications for other systems that are engineered to interact with Web counterparts. Therefore, a method was needed to filter out those spurious results from the selected context.
The next step of filtering was cutting out all projects without tag releases. As it just required the repositories with at least one tag release, in order to make comparisons and calculations with the master branch, therefore, in the previous result repositories that don't contain tag releases should be pulled off. so the calculation metrics are between the tag releases of a developing project in its life cycle.
For removing the repositories without tag releases the script of Git Tag list should be used in the cloned repositories and if it's not empty we keep it for the next filter. But since cloning all repositories was impossible, with the first search result that was just based on the web applications, there is a keyword with the name **"tags-url"** in the JSON files. again with the help of JavaScript language if the result of the Github API query for that key contains length, the repository has been saved in another file. And it was repeated for all web results.
About **1813** repositories that contain tag releases were held for the next step.
The final filtering was for extracting repositories that use at least one of the considered testing frameworks inside them. To search for any of the testing tools considered, a GIT Code Search has been performed on the repositories that are part of the context. The names of the tools themselves are evidence of their usage since they are part of include statements that are needed to make them work. Projects were then divided into two sets, according to the tools they featured. For each of the tools, its adoption has been estimated by computing the TA (Tool Adoption) metric. Sets of projects featuring different tools are not necessarily disjoint: it is possible that a repository features more than just one scripted testing tool. [4] listings

```
const fetch = require('node-fetch');
const fs = require('fs');

const requestOptions = {
  method: 'GET',
  headers: {
   "Authorization": "Basic ***"
  },
  redirect: 'follow'
};

fs.readFile('./results/2- with-tags/2019/December-first-repos
    if (err) {
        console.log("File read failed:", err)
        return
    }
    const repos = JSON.parse(jsonString);
    let finalJson = [];

    await Promise.all(repos.map(async ($repo) => {
        try{
            const res = await fetch(`${$repo.html_url}/searc
            const text = await res.text();
            const htmlRes = JSON.stringify(text);
            const counter = htmlRes.substring(htmlRes.indexO
            if (parseInt(counter) > 0) {
                finalJson.push($repo)
            }
            }catch (err) {
                console.log(err);
            }
            return finalJson;

        })).then((finalJson) => {
        if(finalJson.length){
            const sanitized = finalJson.reduce((acc, curr) =>
                const x = acc.find(item => item.id === curr.
                return x ? acc : [...acc, curr];
```

```
                    }, [])
                    console.log(sanitized[0])
                    fs.writeFile('./results/Ranorex/2019/December-fi
                        if (err) return console.log('Failed to save
                        console.log('File saved');
                        });
                }
            })
        })
```

For each test file the lines of code are counted with the use of the cloc bash tool and contribute to the computation of the Size metrics defined to answer RQ1. TTL (Total Tool LOCs) has been computed for each project, on the master release. As discussed earlier, the use of the git tag command also allows obtaining the NTR (Number of Tagged Releases) metric for any of the considered projects.
Finally, to see which of the considered projects were modified recently, the GitHub Stats API has been used. In particular, the request "GET /repos/:owner/:repo/stats/commit activity" returns the commit activity of last year, giving the number of total commits per week; summing the values over the year gives as result the total number of commits performed during last year (different time intervals could also be considered by taking into account only the values for a number of most recent weeks). Thus, projects with a value different from 0 were tagged as part of the subsets of "alive" projects. The data extraction procedure from GitHub has been completed between September and December 2018.

3.3.2 Test LOCs Analysis (RQ2)

In the exploration of the history of Web application repositories, the versions that have been considered for tracking the evolution of test files are the tagged points of release histories. In addition to those, that can be extracted using the git tag command, the current master branches of projects have been considered, as the last updates of the repositories with which the last code comparisons are performed. The work through **RQ2**, was dependent on each tagged release. The total amount of modified **LOCs** for all projects in their master branch was computed. After that also the total amount of modified **LOCs** for each test tool was computed. The command for checking all differences between each tag is *git diff. [4]*

To answer RQ2, for each pair of consecutive versions of the selected projects, the git diff command has been executed on the whole repository to obtain the total amount of LOCs changed with respect to the previous release. Then, the git diff command has been used again to obtain the number of LOCs added, removed, or modified for each. The values extracted this way allowed us to compute TLR (Tool LOCs Ratio), MTLR (Modified Tool LOCs Ratio), MRTL (Modified Relative Tool LOCs) and TMR (Tool Modification Relevance Ratio) for each test tagged release of any project. [4]
Then, global average values have been computed on the whole lifespans of the projects, using the formulas

$\overline{TLR} = Avg_i\{TLR_i\}$,

$\overline{MTLR} = Avgi\{MTLR_i\}$,

$\overline{MRTL} = Avg_i\{MRTL_i\}$,

$\overline{TMR} = Avg_i\{TMR_i\}$ *[4] with* $i \subset <[1, NTR]$ *being NTR the number of tagged releases featured by the project.*

The preceding step helped for computation of ***TLR***, ***MTLR*** *and* ***MRTL***. *At the end of the exploration of project history, global averages of all metrics were computed:*

NTR is the number of tagged releases featured by the project. [4]

3.4 Result And Discussions

In the following paragraphs, the results which obtained by applying the described procedure are reported. Each of the following subsections concerns one of the two research questions we defined. The results measured for the metrics defined in the section are detailed, along with the conclusions we can base on them.

The detailed measurements extracted for all the examined projects have been published as a data set 100 hosted on FigShare. For each testing tool, we have created two different JSON files, one pertaining to all releases of each project (containing their amount of production code, test code, and modified lines, files, and methods) and one pertaining to all files of each project, and their evolution throughout the release history. The average values that are computed – as explained in the Procedure section – are based on this raw data.

Table 3.3. Characteristics of the selected Testing Frameworks

Number of projects and TA per testing tool.				
Tool	Total	Releases	TA	
Web apps	142829	-	-	[4]
Selenium	150	32444	0.10502%	
Cypress	49	37143	0.03430%	

Table 3.4. NTR, NTC, TTL, TLR per testing tool: average and median (in parentheses) values for master

NTR, TTL, TLR per testing tool: average and median (in parentheses) values for master			
Tool	NTR	TTL	TLR
Selenium	85 (67)	208 (550)	13.537%
Cypress	112 (381)	10372 (118468)	2.6%
Average	98	5290	8.09%

[4]

3.4.1 RQ1: Adoption

*initially are gathered a total of **142829** GitHub repositories featuring the term Web or Web Application in their names, descriptions, or readme files. Then, a significant amount of projects were pruned because of their lack of tagged releases (so they had no history to be investigated). A final set of 199 Web app projects was obtained from 2016 to September 2020 which contain Selenium and Cypress.*
in tables 3.3 and 3.4 the measures answering the metrics pertaining to RQ1 are shown. A summary of the definitions of the metrics is given in table 3.5. The columns of table 3.4 show, respectively: the total number of projects featuring each of the two tools considered; the number of projects featuring at least one tagged release; the Tool Adoption (i.e., TA) metric, computed for each of the selected testing frameworks. Table 3.4 shows the average and median values for the Number of Tagged Releases (NTR), Total Tool LOCs (TTL), and Tool LOCs Ratio (TLR), computed on the sets of projects featuring each testing tool, for their master release. The last row in table 3.4 shows average values for all the considered projects, weighted by the number of projects for each set. [4]
Considering the overestimation due to possible overlaps (since a single project

Table 3.5. Metrics pertaining RQ1

Metrics pertaining RQ1	
Name	Explanation
TA	Tool Adoption
NTR	Number of Tagged Release
TTL	Total Tool LOCs

[4]

can feature multiple testing tools, hence the sets for the individual tools are not necessarily disjoint) about 0.139% of the projects feature tests belonging to one of the two selected tools. None of the testing frameworks reached by itself an important level of adoption in the considered set of Web Application open-source projects.
Even though the total number of Web application projects extracted can take into account some projects that are not likely to feature test tools (e.g. experiments, duplicates, exercises, prototypes, and projects that are abandoned at very early stages) the measures computed for the metric TA give evidence of the lack of extensive usage of scripted automated testing on open-source Web app projects. Anyhow, it is worth highlighting that the study we performed is limited to the testing tools we considered, i.e. it is possible that different scripted testing tools are used by some other projects of the context. [4]
The average and a median number of test files in the sets of projects can be quite small.
Considering the overestimation due to possible overlaps (since a single project can feature multiple testing tools, hence the sets for the individual tools are not necessarily disjoint) of the project's feature tests belonging to one of the two selected tools. None of the testing frameworks reached by itself an important level of adoption in the considered set of Web application open-source projects. [4]
In particular, the absolute number of projects featuring Selenium and Cypress test cases, respectively 140 and 59, is practically irrelevant. Also, other tools like Ranorex, Sikuli, and Watir were considered but no open-source projects have been found on GitHub.
Even though the total number of Web App projects extracted can take into account some projects that are not likely to feature test files (e.g. experiments, duplicates, exercises, prototypes, and projects that are abandoned at very early stages) the measures computed for the metric TA give evidence of the lack of extensive usage of scripted automated testing on open-source Web

App projects. Anyhow, it is worth highlighting that the study we performed is limited to the testing tools we considered, i.e. it is possible that different scripted testing tools are used by some other projects of the context. [4]
The average and a median number of test files in the sets of projects can be quite zero (e.g., for Watir, Ranorex, and Sikuli tools), in which – usually – one test file is written specifically for each Activity featured by the application. In most applications, this is particularly true in the case of small and even experimental open-source projects. [4]
Average TTL and TLR values are very large for the sets of projects featuring both Selenium and Cypress; however, such result is heavily influenced by the small size of the sets (respectively, 23 and 20 projects) and by the presence of the full Selenium framework for Web applications, counting 172,126 tool LOCs of Selenium and counting 490,948 tool LOCs for Cypress. The influence of those individual projects on the average values is confirmed by the largely smaller corresponding median values. [4]
The fact that the set of projects featuring Selenium has the lowest average TTL can be explained by the following reasons: (i) using a white-box testing technique allows to exercise of the functionalities of the application with little coding effort; (ii) the framework is not accessible even for non-experienced developers, and its usage is encouraged by Web applications, leading it to be used also in very small projects, in tryouts, and even for experimental and partial coverage of applications use cases. [4]
The different size of the projects in which Selenium and Cypress are typically used is confirmed by the close average TLR values the sets of projects have, while the respective average TTL values are very different (with the TTL computed for Cypress nearly two times as big as the one computed for Selenium). Slightly bigger test files.

3.4.2 RQ2 - Evolution

Table 3.6 shows the statistics collected about the average evolution of test code, for the two selected testing frameworks. A summary of the definitions of the metrics is given in table 3.7. For every set, TLR, MTLR, MRTL, TMR, and MMR have been averaged on all the projects. The values in the last row are obtained as averages of the two values above, weighted by the size of the two sets. [4]
The values reported for the average Tool LOCs Ratio (TLR) show that – when present – the amount of testing code associated with the selected testing frameworks can be an important portion of the project during its life-cycle if

compared to the number of LOCs of production code. The box plots in Figure 3.1 show the distribution of TLR values for the two sets of projects. The average values range from about 10% (for the set of Selenium projects) to 37% (for the set of Cypress projects). The TLR averaged over the releases of applications is typically smaller than the TLR computed for master releases (see table 3.3): this may be attributable to the gradual of construction of test suites, which may be very small or absent in initial releases. [4]
Average Modified Tool LOCs Ratio (MTLR) measures show that typically around 314 of 23 repositories for Selenium and 696 of 20 repositories for Cypress of lines of test code are modified between consecutive releases of the projects featuring the two analyzed automation frameworks. Very small MTLR values were obtained for the projects featuring Cypress. In general, this should be a consequence of bigger test suites, in terms of absolute LOCs, with respect to the ones written with other testing frameworks.

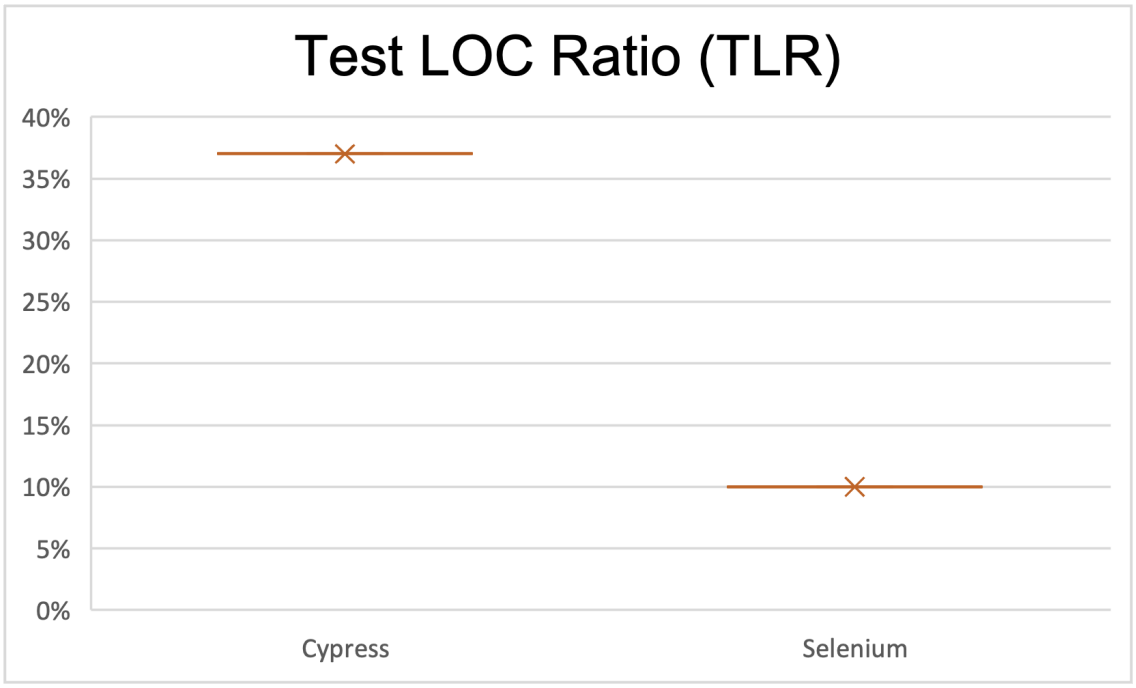

Figure 3.1. Test LOC Ratio (TLR)

[4]

Hence, the influence of a similar amount of absolute modified LOCs would result in a lower MTLR value. The highest value was found for the set of

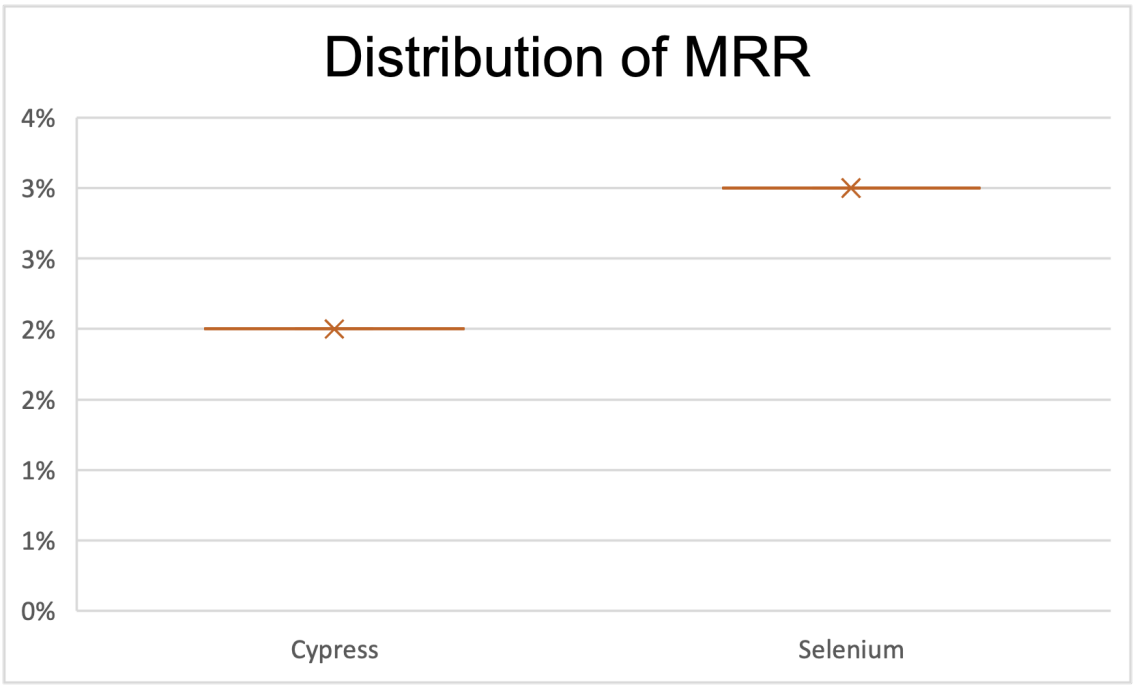

Figure 3.2. Distribution of MRR

[4]

projects featuring Selenium: this can be explained with the very high percentage of total LOCs in files associated with Selenium for these repositories. However, the set of projects featuring Cypress, which also was characterized by a high average TLR, did not exhibit the same trend, having a lower MTLR: this should mean that, even though the important ratio of testing code above project code, few modifications (in both production and test code) were made between subsequent releases on test code associated with Cypress". [4]
The measures about Modified Relative Tool LOCs (MRTL) show that, on average, when the two selected testing frameworks are used, the 57% and 51% of the modified LOCs belong to test files containing code associated with respectively Selenium and Cypress frameworks. With this metric, however, we are still unable to discriminate what is the reason behind the modifications to be performed on test files. The higher MRTL values for the sets of projects featuring Selenium can be justified by the small size of the two sets, and by the nature of the projects examined. For instance, the Selenium framework, on GitHub, is subject to heavy modifications, but in this particular case files that are actually the code of the testing tool should be mistakenly recognized as test code. While the one set of projects featuring code associated with Cypress exhibit close average MTLR values. In general, a higher MRTL should mean minor adaptability of a testing tool to modifications performed on the production code, with more changes needed by the test code as a consequence of changes in the production code. [4]
The mean values of Tool Modification Relevance Ratio (TMR) stayed in the range between 0 and 420591 for big-sized sets of projects, with lower values for the sets featuring both Selenium and Cypress. In general, those values imply that the amount of churn needed for the code associated with a specific testing framework is not linear with the relative amount (with respect to total production LOCs) of such code inside the application: in our case, on average, the ratio between the intervention on test code and the intervention on all production code is about 3/4 of the ratio between test and all production code. The higher TMR value for Selenium is due to some projects in which TLR is rather small, and where in some releases all modified LOCs belong to test files (thus leading to MRTL values very close to 1). [4]

The Modified Releases Ratio (MRR) metric gives an indication of how often the developers had to modify any of the files associated with the considered testing frameworks when they published new releases of their projects. Box plots in figure 3.2 show the distribution of MRR for the projects of the considered context. On average, 2.549% of releases needed modifications in

Table 3.6. Measures of RQ2 - the evolution of test code (averages on the sets of repositories)

Tool	$\overline{TLR}$	$\overline{MTLR}$	$\overline{MRTL}$	$\overline{TMR}$	$\overline{MRR}$
Selenium	36.53	18.16	10.25	4.3	3.265
Cypress	9.42	29.01	133.39	5	1.833
Average	22.975	23.585	71.82	4.65	2.549

[4]

Table 3.7. Metrics pertaining RQ2

Name	Explanation
TLR	Tool LOCs Ratio
MTLR	Modified Tool LOCs Ratio
MRTL	Modified Relative LOCs Ratio
TMR	Tool Modification Relevance Ratio
MRR	Modified Releases Ratio

[4]

any of the test files (with a maximum of 3.265% for the set of projects featuring Selenium, and a minimum of 1.833% for the set of projects featuring Cypress). Since releases may be frequent and numerous for GitHub projects, this result explains that the need for updating test files is frequent for Web developers that are leveraging the analyzed testing frameworks. [4]
It is however worth underlining that those two comparison metrics cannot serve as a precise estimation of the relative importance and needed effort for different typologies of testing: as discussed in more detail in the Threats to Validity section, an exact comparison between two testing frameworks is never possible even if they are based on the same coding language. [4]
We computed a set of evolution metrics on different subsets of the full context of applications. [4]
Projects with at least 1000 LOCs of code associated with Cypress tend to have a smaller TLR value with respect to the full set of projects. This can be evidence of the fact that there is no linear link between the total amount of production and test code, meaning that test suites tend to be smaller, if compared to production code, for larger projects. This trend is confirmed by all the other sets of projects. Furthermore, low TLR values may suggest that the testing code in the selected projects provides only partial coverage of the production code: in such case, it is reasonable that there is not an exact mapping between the amount of production code and test code, with a divergence

that becomes bigger with the size of the application. No relevant differences are found for the other evolution metrics (with the obvious exception of TMR, which depends on the TLR value). [4]
In general, we suppose that every testing framework brings a constant overhead of LOCs, which makes the TLR metric bigger for small projects.

Table 3.8. Percentage of projects without modifications in test files

Tool	Unmodified File
Selenium	45%
Cypress	56.52%

[4]

It must also be considered that the averages reported are heavily lowered by those projects in which files associated with the analyzed testing frameworks are inserted – at the beginning or at some point in their history – but are never modified later. [4]
In table 3.8 we show: the percentage of projects whose test files associated with a given testing framework are never modified; the percentage of projects with no modifications in files associated with a given testing framework (i.e., only additions and modifications of test classes are performed). For instance, in the case of the set of projects featuring Selenium, 55% out of the 23 projects have modified test files between consecutive tagged releases. [4] The graph in fig. 3.3 shows, the ascending MRR measure for all the projects of the context (regardless of the specific testing tools they feature). It is evident from the graph that almost half of the test files are never modified during the lifespan of the project they belong to; this supports our assumptions that test files are often not utilized or abandoned. These results may suggest that the test code associated with the studied testing frameworks is subject to a certain level of aging.[16] With the static analysis of code that we have performed in this work, however, we cannot discriminate between test files that are not modified because they do not need to and those that are not modified because they are no longer utilized by the developers, but are not removed from the project. [4]

3.5 THREATS TO VALIDITY

Threats to internal validity. We have identified the following threats to the validity of our conclusions:

- *The test file identification process is based on some keywords specific to*

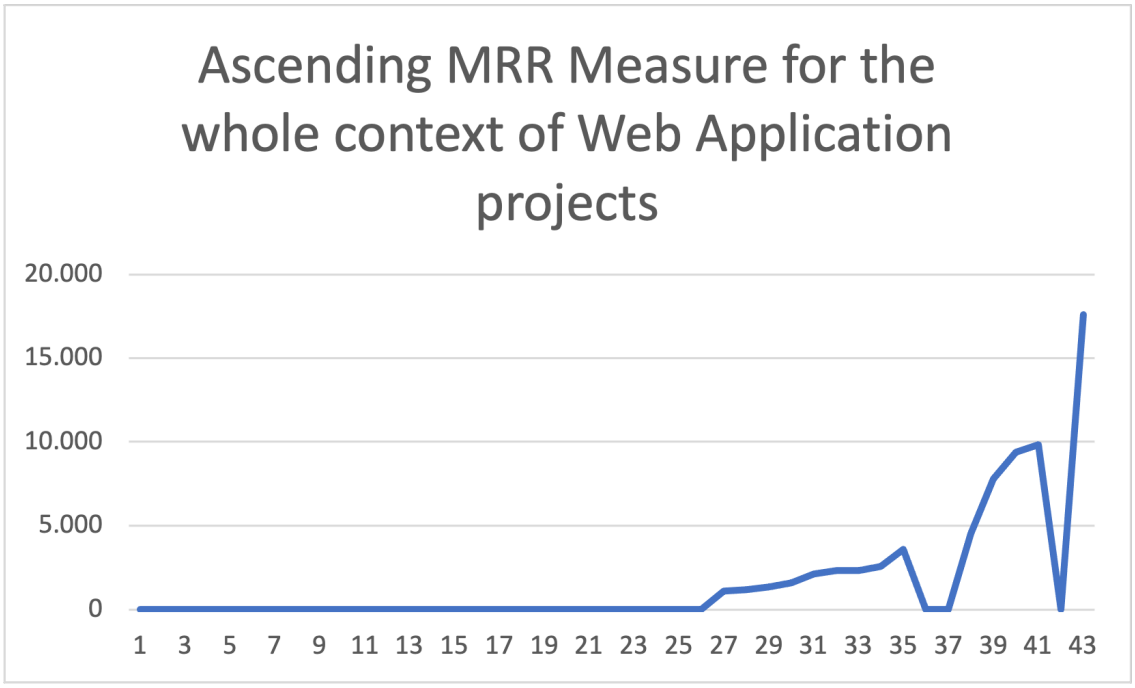

Figure 3.3. Ascending MRR Measure for the whole context of Web Application projects

[4]

each testing tool: any file containing one of such keywords is considered as a test file without further inspection. This procedure may miss some test classes, or consider a file as a test file mistakenly.

- *The number of tagged releases is used as a criterion to identify a project as worthy to be considered for our investigations; it is not assured that this check is the most dependable one for pruning negligible projects.*

- *The metrics we defined have not been tested outside the scope of this study, hence we can not ensure the correctness of the assumptions we based on them.*

- *Structure provided coverage, and the quality of the developed test cases have not been controlled and taken into account by the automated procedure for computing the metrics. Hence, the effects that low-quality tests have on maintenance efforts are not taken into consideration in the discussion we provide.*

- *The performed study was purely static, i.e. test files were not executed to understand whether they were actually working even though they were not subject to modifications. This threat may add biases to the number of test files that we provided as results since they do not consider test code that should be modified because of changes in the AUT or its code but is abandoned by developers. Threats to external validity. We identify the following threats to the generalized of our work:*

- *Testing tools and techniques adopted by relevant industrial practitioners may vary significantly from the ones discussed in this work, and by the related ones discussed in earlier sections. It is not assured that our findings, based on a very large repository of open-source projects, can be applicable to the development of commercial projects.*

- *Our findings are based only on the GitHub open-source project repository. Even though it is a very large repository, it is not assured that such findings can be generalized to closed-source Web applications, or to those taken from different repositories.*

- *We have collected measures for just two scripted automated testing tools. It is not certain that such a selection of tools is representative of other categories of testing tools or even different tools of the same category, which may exhibit different trends throughout the history of their AUT. [4]*

3.6 CONCLUSIONS

In this work, we aimed at taking a snapshot of the usage of automated testing frameworks among Web application open-source projects. We quantified the use of a set of two tools that can be used for testing and that are cited in available literature – Selenium and Cypress – in the projects hosted by the GitHub portal. [4]

We found that the level of adoption of the considered testing frameworks among Web application projects hosted on GitHub is very high. The whole adoption of all the two testing tools considered is about 1.26% of all projects that have a release history. For what concerns individual projects. [4]
Concerning the evolution of test code, on average nearly 8.9% of the total changed lines, between consecutive releases of the same project, belong to the code associated with the selected test frameworks. Such a percentage is quite low if it is considered that code churn is inevitable during the evolution of an application, and tests must adapt to changing requirements or any kind of change in the AUT. However, the average amount of changed lines may also reflect a relatively small coverage provided by test files developed with the studied tools, as it may be suggested by the average relevance of testing code among total production code. Albeit a linear correlation between code churn and man hours in updating code is difficult to proven, this ratio can be considered as a preliminary indication of the amount of effort that developers must spend to keep their test files up to date with modifications that are performed on production code. [4]
In addition to that, when test code associated with two testing frameworks is present, most of the time more code churn is needed in keeping up to date with the latter (with a ratio of even 5 to 1 for the considered tools). A higher maintenance cost for code related to testing was expected: testing frameworks allow to perform system-level testing to interact with the AUT from the level of abstraction presented to the user and are affected by modifications performed on any level of abstraction of the application functionalities. Hence, it is reasonable that tests require more maintenance during the normal evolution of a web application, which is – by nature – typically subject to the rapid evolution of its testing. [4]
These results, however, confirm that – as it is deduced by existing surveys among open-source developers[15]– maintaining a test file is a rather complex and time-consuming task, that can make open-source developers neglect

to test at all, or abandon test code – without making it evolve with the application – after it has been written. [4]
Further empirical studies in this field may directly observe open-source as well as industry practitioners, in order to quantitatively measure their effort in keeping test code aligned with the evolution of the apps and their tests, e.g. in terms of man-hours per release. Furthermore, dynamic evaluations can be performed to quantify the amount of non-working test code kept by developers in their project without performing maintenance on it and to evaluate the way developers cope with aging test code. [4]

Bibliography

[1] Monika Sharma, Rigzin Angmo, *Web based Automation Testing and Tools*, International Journal of Computer Science and Information Technologies, 2014.

[2] Macario Polo, Pedro Reales, Mario Piattini. *Computing Test Automation*, IEEE Software, VOL. 30, NO. 1, January, 2013.

[3] Fei Wang, Wencai Du, *A Test Automation Framework Based on WEB*, IEEE/ACIS 11th International Conference on Computer and Information Science, 2012.

[4] R. Coppola, M. Morisio and M. Torchiano, "Mobile GUI Testing Fragility: A Study on Open-Source Android Applications," in IEEE Transactions on Reliability, vol. 68, no. 1, pp. 67-90, March 2019, doi: 10.1109/TR.2018.2869227.

[5] Junyi Wang, Xiaoying Bai, Haoran Ma, Linyi Li, Zhicheng, *Cloud API Testing*, 10th IEEE International Conference on Software Testing, 2017.

[6] Galin, Daniel, John Wiley Sons. *Software Quality - Concepts and Practic*, 2018.

[7] William E. Lewis, *Software Testing and Continuous Quality Improvement*, ISBN 0-8493-2524-2 (alk. paper)

[8] Thomas Stober, Uwe Hansmann, *Best Practices for Large Software Development Projects*, Springer Heidelberg Dordrecht London New York, 2010

[9] . P. Seth, O. Taipale and K. Smolander, "Organizational and customer related challenges of software testing: An empirical study in 11 software companies," 2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS), Marrakech, 2014, pp. 1-12, doi: 10.1109/RCIS.2014.6861031.

[10] , J. Katharina, T. Matthias, and F. Houdek, "Poster: Challenges

with Automotive Test Case Specifications," 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, 2018, pp. 131-132.

[11] , R. Ramler and J. Gmeiner, "Practical Challenges in Test Environment Management," 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, Cleveland, OH, 2014, pp. 358-359, doi: 10.1109/ICSTW.2014.41.

[12] , M. Leotta, D. Clerissi, F. Ricca and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, 2013, pp. 272-281, doi: 10.1109/WCRE.2013.6671302.

[13] , B. W. Boehm, "A spiral model of software development and enhancement," in Computer, vol. 21, no. 5, pp. 61-72, May 1988, doi: 10.1109/2.59.

[14] , V. Garousi and M. Felderer, "Developing, Verifying, and Maintaining High-Quality Automated Test Scripts," in IEEE Software, vol. 33, no. 3, pp. 68-75, May-June 2016, doi: 10.1109/MS.2016.30.

[15] , M. Linares-Va squez, K. Moran, and D. Felderer, Poshyvanyk, “Continuous, evolutionary and large-scale: A new per- spective for automated mobile app testing,” in Software Maintenance and Evolution (ICSME), 2017 IEEE Inter- national Conference on. IEEE, 2017, pp. 399–410.

[16] , R. Feldt, “Do system test cases grow old?” in Software Testing, Verification, and Validation (ICST), 2014 IEEE Seventh International Conference on. IEEE, 2014, pp. 343–352.