POLITECNICO DI TORINO

Master's Degree in Software Engineering

Master's Degree Thesis

Gamification for Web Testing: Development of a Browser Tool

Supervisors

Prof. Luca ARDITO

Doct. Riccardo COPPOLA

Doct. Tommaso FULCINI

Candidate

Giacomo GARACCIONE

Academic Year 2021/2022

Summary

In software engineering, testing is a vital part of the entire development cycle, and Graphical User Interface Testing is a fundamental step of it, although often neglected due to the fragility of test cases needed for automated testing tools and the fact that such test cases have to be manually created by the testers, which is a cumbersome and unappealing task.

The thesis work aims to find a possible solution to the problem of GUI testing being a tiresome activity by adopting the use of Gamification (the use of elements and strategies typically part of game design in non-recreational contexts).

This choice was made after hypothesizing that performing a testing session of a website while having interesting game mechanics would make testing a more pleasurable activity.

The idea was implemented through the development of a Google Chrome extension that records actions performed by a tester on a website, keeps track of scores and progresses and generates automatic test cases that are compatible with tools that are usually used in GUI testing, aiming to solve the issue of test cases being cumbersome to write manually.

The game mechanics adopted during the development of the extension were selected in order to increase participation and competition; such mechanics are some of the most commonly used ones and include: progress bars detailing page coverage, leaderboards ranking all the users, avatars which can be unlocked and shown, unlockable achievements after performing relevant milestones.

A preliminary usability evaluation has been conducted after development, in order to assess the general usability of the system, to see if there's room for improvement or necessary fixes, as well as to evaluate the reception of gamification elements: the main goal was to observe which of the adopted ones were effective in increasing user participation and interest in the testing activity.

The results of the evaluation showed that the majority of the participants appreciated the competitive aspect of the extension, explaining that the presence of a leaderboard where they could see the scores of other users drove them to interact more with the tested pages in order to increase their scores.

Many evaluators also revealed the usefulness of the progress bars and of a

functionality that marked elements that could be interacted in order to understand which were the missing components to interact with in order to fully test a single web page.

Even though the sample size of the evaluation was too small to be considered significant, it can be assumed that the use of gamified mechanics in the context of GUI Testing can bring positive results, especially in terms of participation and interest, the two main observable obstacles in said discipline.

These results are in line with the principal findings in related software engineering literature and encourage further developments and evaluations of the proposed tool.

Acknowledgements

I would like to express my thanks to my Supervisors, professor Luca Ardito and Doctors Riccardo Coppola and Tommaso Fulcini: their constant assistance and support during the entire work process behind this thesis work has been a great motivating force in my efforts. The amount of trust they showed in my work since the early parts of development encouraged me and made me believe in myself and in the quality of the software I was developing. Then, I would like to thank my family, especially my mother, father and two sisters: thank you for your constant support and presence during not just the thesis work but also during my entire university career as well. Finally, I want to thank all my friends: I cannot list all of you here, but I want to thank you for having given me many wonderful memories during all these years, and I hope there will be a lot more of them in the future.

Table of Contents

List of Tables

List of Figures

Acronyms

API
: Application Programming Interface

CSS
: Cascading Style Sheets

DAO
: Data Access Object

GUI
: Graphical User Interface

href
: hypertext reference

HTML
: HyperText Markup Language

JSON
: JavaScript Object Notation

PC
: Personal Computer

SQL
: Structured Query Language

UI
: User Interface

URL

Uniform Resource Locator

Chapter 1

Introduction

One of the most relevant parts, if not the most relevant, of Software Engineering is the concept of software testing, which consists in the act of examining the behavior of software by means of validation and verification. Testing is an activity that should be performed at every step of the development process (requirements definition, design, implementation, maintenance) but is actually often neglected until the actual deployment of the system, where issues are found, causing losses and problems such as program crashes due to unidentified bugs, constant need to keep applications updated, economic damage and security risks (both informatic security and also real-life security, i.e. planes, automatic vehicles). The reasoning behind testing being usually ignored is that it is mainly a boring and repetitive activity, and this makes it hard to approach by developers. One peculiar case of testing is the concept of GUI testing, which consists in testing a product's graphical user interface to ensure it meets its specifications through a variety of test cases. This work will focus on discussing the development of a browser plugin whose goal is to improve GUI testing and make it more accessible and interesting through the use of Gamification mechanics.

This thesis work will explain how the problem was faced and the solution that was implemented, with the following structure:

- **Chapter 2**. A background research on the theory aspects behind the thesis, touching the fields of Software Testing, with a focus on End-to-End and GUI Testing, and Gamification, with an explanation of its most used mechanics and a study on examples of its applications in software engineering, teaching and software testing;

- **Chapter 3**. A description of the implemented solution, focusing on the adopted technologies, the gamification elements implemented and the additional features offered by the plugin;

- **Chapter 4**. A description of the preliminary usability evaluation conducted on the plugin, with the goal of assessing how the gamification mechanics were received by potential users, along with an analysis of the results inferred;
- **Chapter 5**. An analysis of the limitations found during development and evaluation, as well as a reflection on possible changes that could be done in future works.

Chapter 2

Background

This chapter will contain an explanation of the two main concepts behind the choices made during the development of the plugin, those being Software Testing, with a focus on End to End and GUI testing, and Gamification, with an analysis on its possible mechanics and notable uses in the field of Software Engineering.

2.1 Software Testing

The concept of testing, as already mentioned back in chapter 1, is a vital part of the software development process, as it aims to reduce the number of software failures (execution events where the software behaves in a wrong and unexpected way) by finding and correcting faults, which are the causes behind a software failure and can be generated during the design phase (incorrect, misunderstood or badly explained requirements) or the development phase (software bugs). Software testing, in particular, is a dynamic activity that consists in operating a system/component under specified conditions and observing the results to detect the differences between the expected behavior, human-defined and based on software specifications and requirements, and the one actually obtained; its goal is not demonstrating that a system is perfect and without faults but actually reaching a good enough level of confidence in the system. Software testing is performed many times during development at different levels: unit testing of independent modules, integration testing of interconnected modules, system testing of all modules, mutation testing of program changes, GUI testing; in regards to web applications this chapter, as well as the entire work, focuses mostly on End to End and GUI testing.

2.1.1 End to End Testing

End to End testing consists in testing an entire software product from the beginning to the end of its intended usage flow to ensure that the product behaves as expected during the entire flow without having failures that ruin the user experience. E2E testing's main focus is not on the visible User Interface but on the underlying data which allow the visible system to work and change (i.e. testing a sign-in form by ensuring that the user exists and feedback is given accordingly); the goal is testing how front-end and back-end interact with each other during the execution flow by reproducing how an actual user of the system would behave while using it and checking at every step that data is handled correctly.

2.1.2 GUI Testing

Graphical User Interface testing is the process of testing a software's own GUI in order to ensure it meets specifications and requirements; such process is done through test cases that emulate the behavior a user would have when using the system (i.e. clicking on a button, then waiting for an image to become visible, then clicking on said image, then filling a text field). The creation of test cases is the main problem of GUI testing, for the following reasons:

- The number of operations that can be performed by a GUI can be large and reach unfeasible levels, making testing all possible operations a long and strenuous task;
- GUI operations may also be performed in sequence, which in turn means that defining test cases has to take into account such behavior, increasing the load and challenge for the tester that has to manually define the test cases;
- Regression testing (the act of re-running tests after changes to ensure that software tested previously still performs correctly) can prove to be quite difficult, as a test case designed with a specific position and look of widgets may fail after changes in the GUI.

GUI testing can adopt two possible strategies in order to define and execute test cases:

- **Capture and Replay**. A system where the GUI screen (or part of it) is captured as a screenshot at various times during system testing, with test cases being used in order to replay in an automated sequence the actions executed by finding a match with the screenshot and interacting with the actual screen. Test validation can then be done easily by comparing the expected result (the screenshot taken previously) with the actual content on screen, declaring a passing case if the two are equal. Capture and Replay, however, suffers

a limitation in the form of screens being different and mutable while the underlying state and login remain the same (i.e. changing the position and the color of a button while leaving its behavior untouched would cause a test case built on the previous situation to fail); even something as small as a font or color change in a text may cause a failing test case even though a user could easily understand that the end result is the same. Another issue that comes from test cases being based on screenshots is that different devices can have different sizes and resolutions, causing a test case that is perfectly working as intended on one device to fail on another simply because a widget has a size difference while the logic underneath is the same, and already tested;

- **Event Capture**. A strategy where GUI interaction data is captured based on events and thus disconnected from the actual appearance of the system (i.e. record the click on the button with *idButton=b1* rather than record the click on the button with a specific look and position). Generation of test cases takes into account this situation and performs actions by replaying the events instead of directly replaying clicks.

2.2 Web Testing Technologies

When it comes to testing entire web applications there are some effective tools that can be employed successfully: these tools follow the specifications and rules of GUI testing by adopting *Capture and Replay* strategies.

2.2.1 SikuliX

SikuliX is a Capture & Replay GUI Testing tool based on image-recognition features provided by the OpenCV library[1] designed in order to automate any kinds of activities that can be performed on a PC, and is used as a tool that can perform automated and visual testing, as its main feature is using image recognition techniques in order to identify GUI elements and then the emulation of mouse and keyboard to interact with the identified elements. SikuliX runs as a *Java* based program that launches a dedicated IDE where users can write scripts in *Python* that will then be run by the IDE and perform the actions executed. The IDE employs a screen capture function that allows users to directly specify which elements are to be found and then interacted with; an example of how the screen capture works is shown in figure 2.1: images can be taken with the button *Cattura schermata* or be chosen from ones already present in the user's file system by using *Inserisci immagine*. The script executes a simple operation by waiting 10 seconds for the image shown to become visible through the *wait* command and then, after

said image has been found, clicks on it by executing the *click* command, going to a new page, as can be seen in figure 2.2.

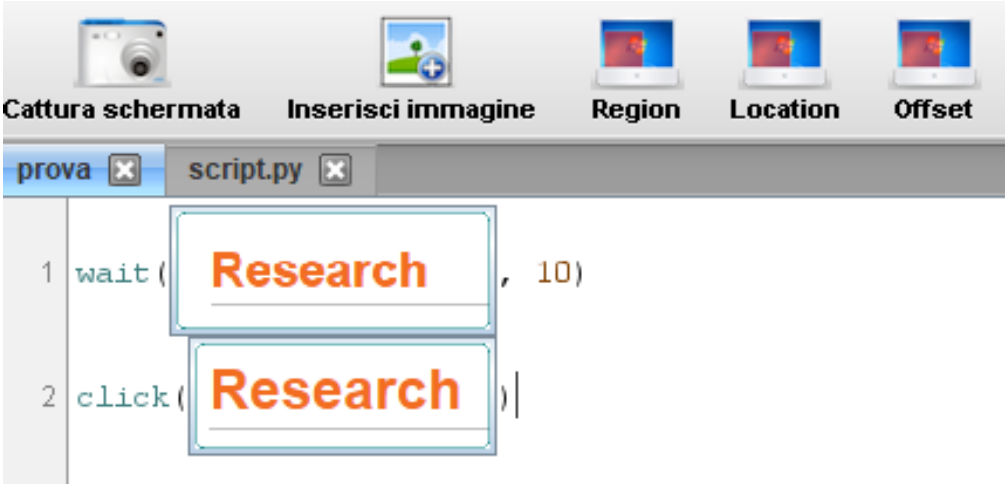


Figure 2.1: SikuliX IDE containing a script for finding and clicking a link

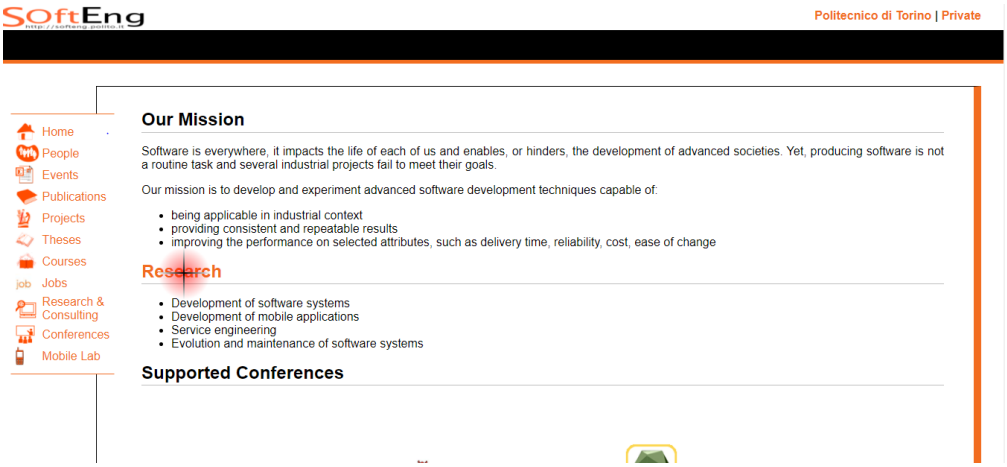


Figure 2.2: SikuliX cursor finding and clicking the specified link

When using SikuliX, however, there are some issues that may come up related to matching images in the script to the content shown on screen:

- multiple elements present in a page to test may match with the same image;
- an image taken through external means may not have a match on screen.

These issues are due to the fact that, by default, the *find* command behind the aforementioned *wait* and *click* methods that looks for images on screen works with

a default minimum similarity of 0.7 (in a range between 0.0 and 0.99), and such value may need tinkering in order to find a match with only the required element. A solution to this issue is given by the *Pattern* class, which provides two relevant methods that can allow greater customization in handling images:

- **similar(similarity)**, which overrides the default similarity with a user-specified value which can be found by working around the IDE's *Pattern settings* shown in figure 2.3, where a lower similarity parameter brings up more elements present in the page that match, although not exactly, the image embedded in the script;
- **targetOffset(dx, dy)**, which changes where the click on the found object will take place, moving from the default position identified as the image's center.

The issue of similarity between an image inserted in a SikuliX script and the content of the page where the screen was taken from will be discussed in detail in the following sections, along with the solutions that have been tried.

2.2.2 Selenium IDE

The Selenium IDE[2] is a Chrome Extension used to record and playback in an automated way operations on a web browser, intended to give users an easy way to generate tests in Capture & Replay mode by being based on Selenium WebDriver, a more powerful tool used in automated and distributed testing that allows the creation of tests even in script mode but that also requires to have programming knowledge, a relevant difference when compared to the IDE, which can be used even with no programming ability whatsoever. The IDE can record actions on the different elements of a web page, identifying them based on different characteristics such as their *HTML* identifier, their *CSS* selector or their *xpath*, a list of nodes that starts from the interacted element and goes all the way up through the *HTML* document until an element possessing an identifier is found; many different actions such as clicking, typing and hovering can be recorded, and the IDE can also perform operations such as running JavaScript code directly in the page. Figure 2.4 shows more in detail how said operation works: the first command in the script opens a new Chrome tab with the URL specified as *target*, while the second command looks for and then automatically performs a click on the only element in the page document that verifies the condition *linkText= Research*. Finally, the *target* dropdown menu displays the different criteria that Selenium adopts to decide which element to interact with: these criteria may be based on position in the *HTML* document related to an element with an identifier, being a *link* element with a given text or a given *href*, which is the target page that the clicked link redirects to.

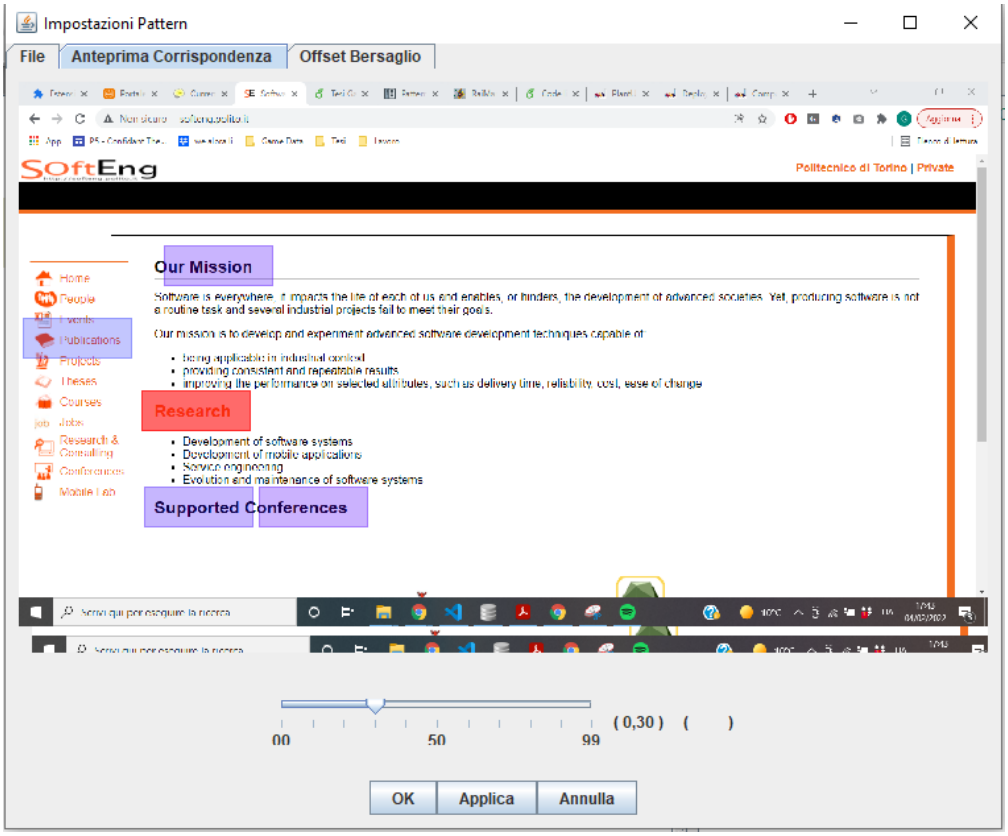


Figure 2.3: SikuliX IDE section for setting an image's similarity parameter

The IDE allows to download a *.side* file representing in a *JSON* format the entire project, which makes a session easily portable and at the same time easily editable, as the file can be opened with any text editor to view the different parameters; the structure is depicted in figure 2.5, in which the *commands* array is presented twice in order to ensure better readability of the figure. The structure of the file shows a division in different elements:

- **project**, the root of the entire structure, identified by its name, has a list of visited URLs with the starting one specified, of test cases and test suites used and a list of optional plugins installed;
- **suite**, one test suite with specified rules such as the ability to run in parallel to other suites, the ability to run multiple test cases with a single browser instance, the timeout to wait for tests to complete before terminating the test run and the identifier of the test case(s) used;
- **test**, a test case identified by a list of one or more commands;

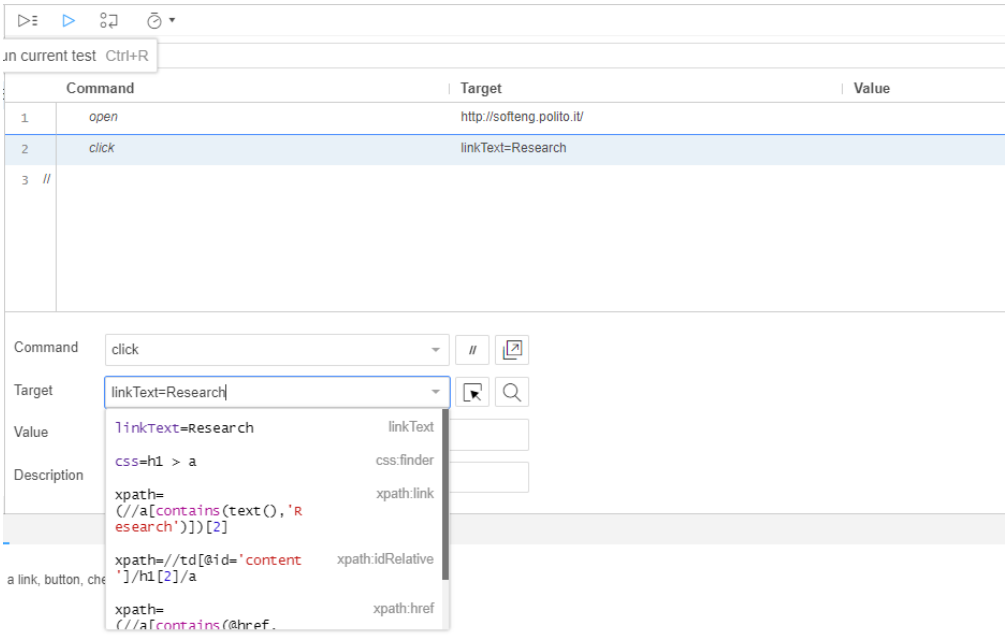


Figure 2.4: Selenium IDE containing a script for opening a web page and clicking a link

- **command**, the most relevant part of the script file, requires specification of the command (*click*, *type*, *hover*, *execute script* are some of the possible options available, a target which identifies univocally one element on the page (identifier, *CSS* selector, *xpath*, text of a link, destination page). An optional value can also be specified for input fields, and multiple targets can also be listed, with Selenium trying all of them until a correct one is found or all of them fail.

The easy to understand and replicate structure of a *.side* file means that, as long as one knows the *HTML* document structure when an element is interacted with, finding out the selectors required requires low effort and the creation of a file that can replicate actions, as will be discussed later, is a quick and simple process.

2.2.3 Scout

Scout[3] is a Capture & Replay tool that works differently compared to the previously mentioned tools, as it makes use of a recently developed technique known as Augmented Testing[4]. Augmented testing performs testing by applying a new visual layer between the tester and the system under test; this additional layer is called Augmented Layer and retrieves information from the original GUI in order to emulate it and merges it with additional information (actions to perform, results

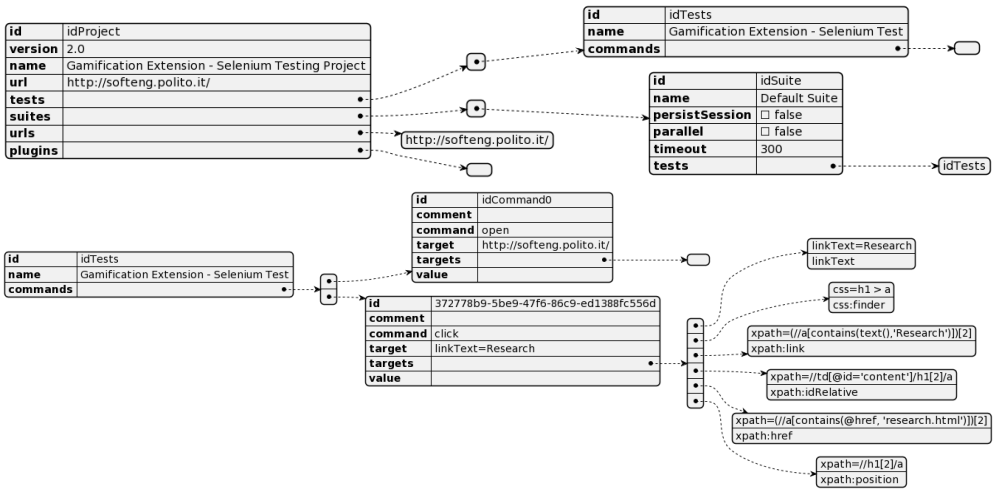


Figure 2.5: Schema depicting the structure of a file read and written by the Selenium IDE

to check, identified issues, comments, statistics) to create the AL the tester can interact with. Actions performed by the tester on the Augmented Layer are recorded and relayed to the system, which then changes accordingly to the input received, causing an update to the Augmented Layer and continuous interaction between it and the tester. A testing session performed with Scout begins by having the tester specify some settings such as the application name, whether it is a desktop or web application, the type of browser for the latter type and the *HomeLocator*, an URL in case of a website or a path to an executable file for a desktop application which is launched when the session starts. The main action performed on the Augmented Layer is the use of color-coded conventions to convey messages related to the state of widgets by applying changes to them; such conventions are reported in table 2.1. The conventions employed by Scout have been a source of inspiration for some

Style Change	**Meaning**
Purple circles	Suggestions (scenarios to try to improve coverage)
Green rectangles	Valid results to check
Red rectangles	Issues
Yellow rectangles	Invalid results or unconfirmed issues
Grey rectangles	Unevaluated results or unevaluated issues

Table 2.1: Scout conventions for widgets

decision choices taken in the development phase of the plugin, as the concept of

changing the style of interacted widgets was thought of as a good way to show users which actions were taken in the past and which were still not undertaken; the concept of issues and their signaling was also explored during development, as it was perceived as a significant feature for a testing tool. Augmented testing enables much faster communication between the tester and the system but has the drawbacks of requiring an increased overhead due to the Augmented Layer being placed between the user and the system and thus requiring additional resources by having to constantly send information back and forth between the two and the risk that interaction between the Augmented Layer may be different than when using the actual system.

2.3 Gamification

Gamification is defined as "the use of elements and techniques typical of games and game design in a non-recreational context". Its techniques are often used in working contexts with the goal of trying to involve people in having more fun and feeling more involved into daily activities which might be seen as boring or uninteresting through gaming, its dynamics and mechanisms such as:

- Points to collect;
- Unlockable levels;
- Obtainable gifts or rewards;
- Badges to show.

These mechanics are used in order to stimulate some primary human instincts and desires such as social status, rewards and achievements, spurred by natural competitiveness, with the goal of creating and satiating human needs. A product is gamified if it provides goals/milestones to reach, levels in which to progress, competition with other users, sharing of one's own achievements and earning of goals.

2.3.1 Gamification Mechanics

The implementation of gamification mechanics is a very effective method to involve people in the activities of an application or an offered service since the user quits being just a simple passive consumer of information but becomes active by using the gamified product. With active behavior also comes an improvement regarding the conveyance of a message: by encouraging people to perform some specific action, the message can be connected to the action itself, wrapping everything together in

the same experience. Another advantage comes in the form of obtaining feedback in the form of data, allowing to classify users and understand which are the tastes of each user, allowing a more intensive focus on a specific target population and/or an expansion of the potential user base. At the root of a gamified product stand simple basic mechanics, used to ensure that users can have a stimulating and satisfying experience thanks to effort, interest and participation; the root mechanics are reported as follows:

- **Points/Credits**. Collecting points is a very effective tool in increasing user participation, seeing as users are incentivized to perform specific actions in order to increase their score, with no real value connected to it. At the end of the experience the obtained score may be used to redeem prizes, giving users the sensation of having invested well the time used to earn points;
- **Levels**. A way to classify users according to their earned score, usually reflecting real-life contexts (workplaces, schools, social environments) where people are divided into different classes with distinct hierarchies. A new level always introduces a new goal to reach, generally by obtaining a predetermined score, and access to new privileges which can be shown in the user's profile. This can make it so that the user experiences positive feedback regarding the effort spent in completing the various levels, encouraging him in continuing the experience in a more focused way, all in order to obtain more and more new privileges;
- **Virtual currencies**. They can be won, traded or bought by users according to their score, their role is to encourage users, who will increase their effort in order to obtain a specific reward, which will then be used to customize an optional avatar inside the system, building a well defined and exclusive identity for the user. The rewards may also be obtained in exchange of real-world currency through microtransactions, allowing the creators of the system to earn a profit;
- **Leaderboards**. The most effective way to establish a competition between users, ordering and dividing them into different groups, leaderboards may be based on different kinds of criteria (score, time spent, level, performance). Users are thus encouraged to spend more time interacting with the system, putting more effort in order to increase their position in the leaderboard, keeping track of their score as much as the ones obtained by other users.

These mechanics make up what is used in most cases of gamification applied to different activities, the so-called **PBL Triad**, which is made up of:

1. **Points**. The different scores obtained by the various participants to the gamified activity;

2. **Badges**. Distinctive elements that are given to the participants whenever they reach a certain amount of points or get to a specific, relevant milestone;

3. **Leaderboards**. An ordered list of all the participants based on their score, it encourages competition by appealing to the human desire to prevail.

The PBL Triad has been thought of as a relevant starting point in terms of gamification mechanics to implement, and all the three elements make up a part of the extension in some way (i.e. badges have been implemented as a combination of unlockable avatars that can be obtained by users and displayed to others).

2.4 Gamification in Teaching

Gamification has been adopted in some examples of teaching in fields related to Software Engineering in order to increase student motivation and participation, with hopes that such an approach would yield increased results and improve performance by having students more involved with a more familiar and friendly approach. This section will discuss some examples of gamification used in teaching and its results. One of the first cases of gamification used in the field of teaching is the case of a Software Engineering course[5], where teachers decided to apply gamification to teaching in order to keep student interest and motivation high, with the reasoning that interesting game mechanics drive players to always come back and that such an approach could yield positive results. The teachers used as the basis for their decisions concepts based on psychology, especially the fact that humans are motivated to work on cognitively difficult tasks when they are granted:

- **Autonomy**. The course content was provided to students in a clear and defined way, and students were given freedom in terms of order and speed related to how they intended to approach the studying process;

- **Purpose**. The course had a clear and rewarding goal at its end;

- **Mastery**. Students were free to take exams that could be repeated in order to validate their understanding of the material and keep track of their progress towards the final exam evaluation.

The course was based around three different fields (Software Development, Project Management, and Communication) and was focused on giving students the freedom to choose which ones to tackle first; after having completed two of the three study areas students were eligible to take part in what was called a *Marketplace*, where they could form groups with which they were able to undertake the course projects. Gamification mechanics used during the course were levels (for each separate field, progress in one would unlock the next), points awarded for completing levels and

for helping other students pass their levels, progress bars and leaderboards where students could see their progress in relation to their peers. Unfortunately, results were not as good as the teachers had expected, as the survey performed at the end of the course revealed that the majority (around 70%) of students did not view challenges, autonomy or reaching goals as motivation for playing games; additionally, many students felt that a gamification platform needs, in order to be successful and interesting, to give a good overview of tasks, to give feedback and to have intuitive controls.

Another case of gamification being employed in the field of teaching has been performed by Anderson et al.[6], where a Data Science course saw the use of the **Learn2Mine** platform, an open-source, cloud-based gamified learning system structured around incremental programming assignments in data science. The platform works by splitting the problems offered to students in separate incremental modules, which give immediate feedback, in the form of notifications and badges, when completed; Learn2Mine is also a useful tool from the point of view of instructors, as its immediate feedback allows students to understand easily their errors, meaning that teachers can focus more on observing the general progress rate of the class, with the presence of a dedicated section containing student activities (tasks completed, time for tasks, progress). A strong point of the platform, additionally, is the fact that the gamification mechanics adopted (badges, points, leaderboards, progress bars, etc) are designed to be as less intrusive as possible, in order to allow students that are not interested in gaming to use the system while easily ignoring the gamified elements. The use of Learn2Mine was evaluated at the end of the course through a questionnaire based on Likert scale questions: the results showed that a gamified approach brought a general improvement in terms of exercise submission, as well as an increase in understanding of the course arguments; additionally, students indicated through free comments that both the division of exercises into small sections and the immediate feedback were quite favorable elements of the platform but, at the same time, few students notified the gamification elements compared to the dividing approach; the latter reasoning was attributed by the authors of the study to the unintrusiveness of the gamified elements.

A more recent example of gamification applied to teaching is the case of a Software Engineering course that saw the use of **Code Defenders** [7], a game that employs a division of players in two different groups: attackers that introduce faults over code and defenders that write test with the goal of identifying these faults. Faults created by attackers are called mutants and award points to the defending team if a test that passes normally fails in presence of a mutant, which is thus considered detected, while an undetected mutant awards points to the attacking team, creating an interaction loop where defenders write stronger tests and attackers develop subtler mutants. The course saw the use of CodeDefenders

in practical, weekly sessions of computer lab exercises with one *Java* class to test for each session and the students alternating between roles each week (excluding the first one, where all students played both roles), with all classes barring the first one being taken from real-life open-source projects with all possible references removed in order to avoid cheating; classes were selected in order to make writing tests and mutants increasingly more difficult as the sessions went by and, in order to prevent unfairness and grant equity in terms of challenge, students were divided into groups based on the scores obtained during the previous session, so as to avoid situations where a strong defender would dominate a session if put against less skilled attackers. The experience was judged as overall positive as it yielded positive and valuable results, according to the authors of the study, who evaluated the experience through the following criteria:

- **Student improvement**. Measured in terms of code coverage and mutation scores achieved during the semester, both values saw a general increase as the course went on;
- **Grades**. An analysis performed by authors revealed a moderate correlation between active participation in the experience and final exam scores, observing that, generally, more active players got higher scores;
- **Student feedback**. Students undertook an optional anonymous survey at the end of the experience, and the overall majority of the participants revealed that playing the CodeDefenders game helped them improve and learn more skills, especially when playing the defending role.

2.5 Gamification in Software Engineering

The gamification approach has been tried not only in the field of teaching, but has also seen use in some cases in software engineering; more precisely, both the general development process and the testing process saw the use of gamification strategies in order to improve activities which are usually not viewed as interesting or pleasurable to do, even though they are a relevant part of a development cycle and their absence could cause issues.

One first example of gamification applied to the development process is the case of **CleanGame**[8], a platform developed in order to assist the refactoring process with gamification mechanics. The term *refactoring* refers to the process, performed during software development, of changing the structure of internal code without actually changing its behavior; in order to perform a correct refactoring of code it is necessary, however, to identify and remove the so-called *code smells*, instances of poor design and bad implementation choices which can undermine code maintainability and understandability (first defined by Fowler et al.[9]), as

the removal or correction of code smells is what causes the code improvements expected to obtain when refactoring. The main issue is that refactoring, while being addressed as one of the best practices when developing, is an activity that is rarely taught and enforced, as it is a menial task that can be neglected in favor of focusing more on design or implementation, which are perceived as more relevant tasks. The CleanGame platform is divided into two main modules:

- **Quiz module**. A module that presents various quizzes about code smells based on the definitions made by Fowler et al.[9], it also allows the definition of new questions. Employs gamification mechanics such as player status, scores, timing and ranking of the 10 best scores in a quiz, with a penalization system that has players lose points if they skip a question or take too long to answer it;

- **Code smell identification module**. This module employs a static cross-language code analyzer, **PMD**[10], to create a list of identified code smells found in the *Java* source code. It makes use of tasks to identify code smells and uses the same mechanics as the Quiz module, including the penalty system, which also takes into account the option to ask for hints, related to metrics used to detect a code smell, refactoring aimed to address a code smell, or short definition of a code smell; asking for hints will, however, cause a penalty and reduce the score.

Another strong advantage of CleanGame is that it is fully integrated with the GitHub API: when creating a room in the identification module, the user must provide an URL corresponding to a Java GitHub repository containing the source code to be analyzed; such code is cloned and transformed into an abstract syntax tree in a fully automatic way to create an oracle of smells-related questions usable by the module. Results of the experiment, noted by the authors as being the first case of gamification applied to the process of refactoring, showed that CleanGame brought a general improvement in performance: subjects were able to identify code smells approximately half more effectively when using the platform, with the authors hypothesizing that this was due to the definitions given by the tool, which are absent when using a standard development IDE; additionally, participants remarked the usefulness of some gamification mechanics, mainly the competitiveness, the use of tips and the presence of dynamic leaderboards. The authors of the study, however, remark the fact that the study has been performed only in a limited way, with a small sample of participants, and limiting the analysis only to the *Java* language; they conclude that further experimentation could be needed to safely assume that gamification could bring benefits to the refactoring process.

A recent study, performed by Muñoz et al.[11], analyzed various cases of gamification strategies applied to software engineering with a focus on teamwork, aiming

to identify which mechanics, if any, can be applied to reinforce the creation of highly effective teams for software development. The study revealed that the most common gamification elements were as follows:

- Leaderboards;
- Points system;
- Badges;
- Levels;
- Progress bars.

The two most used elements, leaderboards and points-based systems, show that elements that encourage participation, motivation and interest in people that use a gamified system are quite common and effective; such benefits can be applied in working environments, where the purpose of gamification should be to improve social interaction and collaboration. The study is particularly useful in confirming once again that leaderboards, points and, to a less extent, unlockable elements that can distinguish participants are effective gamification elements that should be considered as main parts of a gamified system.

2.6 Gamification in Software Testing

Reprising what was said back in section 2.1, software testing is, at the same time, a vital part of software engineering but also the part that is most often neglected, leading to problems down the line as software defects come out and affect negatively the user experience. More specifically, the fact that both End-to-End and GUI testing are neglected can cause huge issues, seeing as they consist in testing an almost ready product, at a point in the development cycle where issues have to be detected, as they can probably be connected to large parts of software that have to be fixed. Many solutions could be tried in order to revitalize the testing problem: in terms of GUI testing, for example, being able to go beyond the limits of automated testing (manual generation of test cases is a boring and long activity, test cases are extremely frail and can fail if the system under test sees changes in its user interface) would bring to great improvements. Many studies have tried the gamification approach, with the idea that making the testing activity more interactive and appealing could bring various advantages: the simple use of mechanics aimed at increasing interest and participation coupled with a neglected and boring activity should create an effective combination, turning the testing activity into something that can be done with ease and peace of mind.

One example of gamification elements applied to testing is a study by Costa and Oliveira[12], who theorized the use of gamification as a means to improve the teaching of Exploratory Testing, in what is one of the first examples of game techniques applied to the exploratory testing process. The strategy they theorized consisted in using a pirate-themed setting, focused on treasure hunting, where solving puzzles translated to finding treasure (bugs), increasing one's score. The study employed a vast amount of gamification mechanics, with the most relevant being the following:

- **Profile**. All users involved in the experiment had a specific profile, with three possible profile types:
  - *Tester*. Students involved in the testing activity;
  - *Expert*. Is the driver of the gamification aspect, solves the testers' doubts and analyzes the results;
  - *Judge*. Observes if the testers are performing the correct activities that will award points.
- **Avatar**. Assigned to each student, there are an *Activity Avatar*, obtained in each activity step based on the activities performed by the student, and a *Participatory Action Avatar*, which depends on how much the student actively participates in the class activity. Each avatar is different for each step of the gamified process, there are rewards for reaching the maximum score associated to an avatar and a *Final Avatar* for each activity step is computed, based on the scores of the two avatars; at the end of the entire experience a *General Avatar* is computed, depending on the arithmetic mean of all the past Final Avatars, which expresses the average performance of the student;
- **Activity**. The different actions that a student can take during the different phases, they give rewards when completed. These rewards can be medals or coins, named *bitskulls*, that can be used in order to purchase resources such as cards;
- **Cards**. Special resources that can be equipped to an avatar in order to give various kinds of advantages (defense cards to protect from attacks by enemies, offense cards to attack enemies) or used for cosmetic purposes (avatar customization).

The entire process of exploratory testing would divide students into groups and take place over seven days, with the most relevant activity being an exploratory analysis of the code under test in order to find as many defects as possible (called the **Treasure Hunt** phase) and a definition of which defects had priority over others, producing a *defect report*; teams then entered the **Battle** phase, where they

exchanged their defect reports in order to analyze the reports made by other groups in terms of prioritization of defects, clarity in the definition of the scripts associated to defects and reproducibility of defects, producing an *analysis report*. An expert would then analyze the reports in order to assign points to all the teams, giving recognition to the teams that obtained particularly high scores: these teams were also allowed to obtain an additional reward card after solving correctly a puzzle. At the end of the entire testing endeavor teams that had obtained the highest possible rank were given three riddles to solve, with the solutions pointing to physical locations in the classroom where a treasure was hidden, with the assumption that this would bring to higher immersion in the pirate setting and also an increase in motivation due to the physical artifacts. The results of this kind of application of gamification were defined by the organizers as to be evaluated by comparing the forms taken by the students before the experience, data taken from the exercises and feedback gathered at the end of the experience, all in order to understand whether gamification could be a positive addition to an exploratory testing experience.

Chapter 3

Design and Architecture

The developed plugin runs based on many different kinds of technologies, connected to the different parts that are required for a web-based tool (browser capabilities, data storage); it can also generate scripts that are compatible with two already existing automated testing frameworks, namely Sikuli and Selenium. What follows is a detailed explanation of all the technologies that come together in order to create the plugin.

3.1 Tools Used

3.1.1 Google Chrome Extensions

In order to develop a browser plugin, it is first necessary to understand exactly what is a plugin used by a browser; in this case, the selected web browser was Google Chrome, which defines its plugins as Chrome Extensions[13], which will be the term used from this point on when referring to the plugin. Chrome Extensions are small software programs that are used in order to customize the web browsing experience, letting users tailor Chrome functionality and behavior in many ways, by providing features such as:

- Productivity tools;
- Web page content enrichment;
- Information aggregation;
- Fun and games.

Extensions are built on web technologies such as the *HTML*, *CSS* and *JavaScript* languages, run in a separate and sandboxed environment and interact with the

Chrome browser. Extensions operate by means of an end-user UI and a set of developer APIs that can be used in order to allow the execution's code to access features of the browser such as activating tabs, modifying network requests, starting file downloads and so on.

Manifest File

An extension is identified by a manifest file, a peculiar *JSON* file which specifies information about the extension: the mandatory information to be included consists of name, description and version of the extension, all displayed in Chrome's dedicated page used for the handling of extensions and the version number of the manifest, which is used in order to specify the rules and the features that the extension is able to implement. An example of such manifest file is reported below:

```
{
    "description": "A plugin that enhances app testing with
    gamification strategies",
    "name": "Gamification Plugin",
    "version": "1.0",
    "manifest_version": 3
}
```

The manifest file is also used to list which are the files that make up all the logic and behavior of an extension:

- **service workers**, dedicated scripts that listen for browser events exposed by Chrome's extension APIs in order to react accordingly and enhance the user experience. The latest rules about extension development state that an extension can only have one single service worker declared, with the following format:

```
    "background": {
        "service_worker": "background.js"
    }

```

- **actions**, elements located to the right of the access bar in the browser and that can be divided into browser actions, valid for use on most pages, and page actions, which are intended to be used only on a few, specific pages. The declaration in the manifest file follows the schema:

```
    "action":{
        "default_popup": "popup.html"
    }

```

- **content scripts**, files that run in the context of web pages and can be used in order to implement the logic of the extension and how it can improve the user experience; a relevant feature of content scripts is that they run in an isolated world, a private execution environment that is not accessible to the page or other extensions, making it so that a content script is able to perform operations without conflicting with the page or other extensions' content scripts, with the drawback of having variables and functions declared in an extension's content script be accessible only by that specific extension. What follows is an example of content scripts' declaration, where a mandatory field is used in order to specify the URLs where different content scripts are allowed to run.

```
    "content_scripts": [
        {
            "matches": ["*://*/*"],
            "js": [
                "content_scripts/cs1.js",
                "content_scripts/cs2.js",
                "lib/lib1.js",
                "lib/lib2.js"
            ]
        },
        {
            "matches": ["*://google.it/*"],
            "js": [
                "content_scripts/gcs.js",
                "lib/glib.js"
            ]
        }
    ]

```

The first object in the array lists four different content scripts that can run on all URLs, as specified by the *matches* array, while the same array in the second one specifies two other content scripts that can only run when in a page whose URL belongs to the domain *google.it*; such distinction can be used in order to define different rules and behaviors for the different websites supported by an extension.

The manifest file also has to list all permissions required by the extension, which equates to listing which are the Chrome Extension APIs the extension is allowed to use; some of these permissions allow the extension to access the browser's tab system, know which is the currently active tab, initiate and manipulate downloads and show system notifications to users, as shown in the example below.

```
"permissions": [
        "tabs",
        "storage",
        "activeTab",
        "scripting",
        "downloads",
        "background",
        "notifications",
        "<all_urls>"
    ],
```

Finally, since an extension may have some local resources (i.e. images, *CSS* style files) which cannot be normally accessed by websites or content scripts, these resources have to be declared in the manifest file in the following format:

```
"web_accessible_resources": [{
        "resources": ["img/*"],
        "matches": ["<all_urls>"]
    }]
```

Each object in the array specifies a list of resources and a list of URLs that can access said resources without causing issues.

Chrome Storage

A relevant feature used by the extension in order to store data is the use of Chrome's Storage API, which provides powerful capabilities in terms of data storage, allowing automatic synchronization through *storage.sync* and access to storage from both content scripts and service workers, allowing for easy sharing of data between different functionalities of an extension. Using *storage.sync* is particularly useful as it allows data to be synced to any other Chrome browser where the user has signed in, bringing to shared data even among browsers, and it also is used to sync data after the browser goes offline and comes back online, with the guarantee that data stored while offline is stored locally waiting for the connection to be restored.

3.1.2 Server

Most of the data used and needed by the extension (more specifically, all data related to pages visited, users' profile information and widgets interacted with in pages) cannot simply be stored in Chrome's storage, as the volume of information can grow to extremely high levels as the number of users, pages and widgets increase. An external server is thus deployed in order to have an easy way to access and store the extension's required data.

Database

Persistent data is stored inside an *SQL* database whose function is to keep track of the evolution of everything that users perform while using the extension, with different tables used to store the different information required. Figure 3.1 displays a representation, using the *Entity-Relationship* model, of the different tables in the database, the fields of each table and the relationships that interconnect the different tables. The tables that make up the database all perform a specific

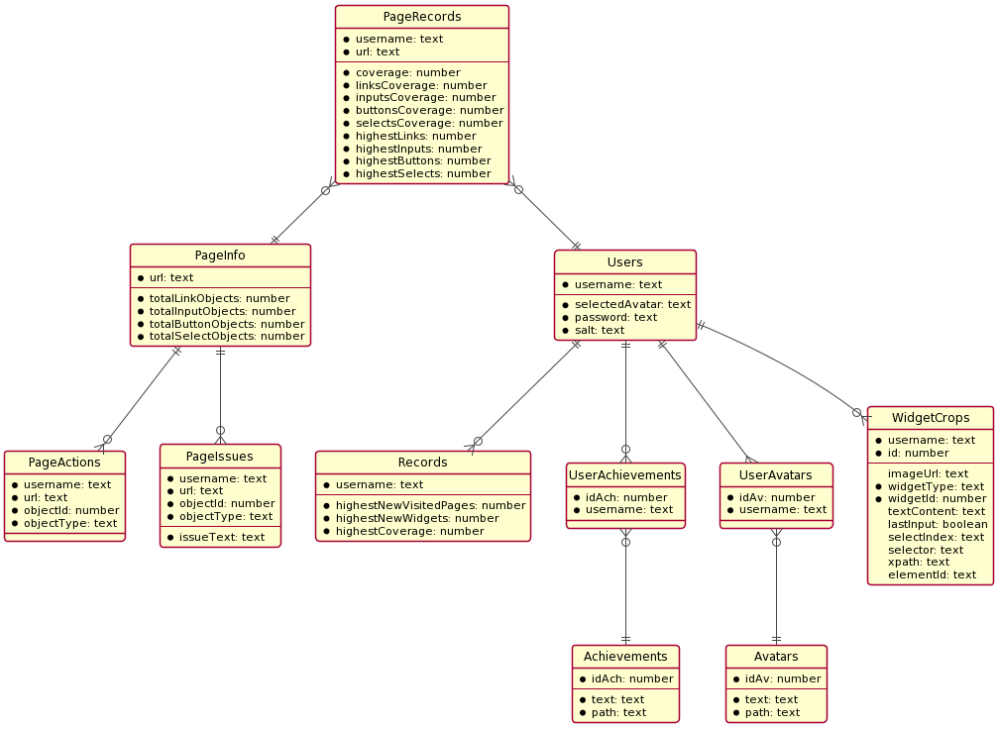


Figure 3.1: Database schema

function; in detail:

- **Achievements**. Contains all the different achievements that can be unlocked

by using the extension, with the text detailing the unlocking criteria and the path to the image associated with each achievement;

- **Avatars**. Following the same reasoning as the **Achievements** table, this table contains the list of all avatars, with a path to the image and a name as parameters;

- **PageActions**. Contains the entire list of actions performed by users on the different pages. Each row keeps track of which user performed an action and on which page, with the action being identified by the type of interacted widget and its index relative to all widgets of its kind in the page;

- **PageInfo**. Contains information about all pages visited at least once while using the extension. Information stored consists of the amount of the different widgets registered by the extension which are present in the page;

- **PageIssues**. Acts in a similar way to **PageActions**, with the difference between the two tables being that this one stores issues reported rather than actions performed and it additionally stores the description of the issue added by the user when reporting;

- **PageRecords**. Stores, for each user, the records obtained in all pages visited by said user during all sessions. Records are different for each page visited and include coverage, both for the single page and for all kinds of widgets, and the count, both for each kind of widget and overall, of the highest number of widgets found in a single session;

- **Records**. A table used to store information about the highest scores obtained by users (new pages found in a session, new widgets found in a session, page coverage). Said scores are used in order to create the leaderboards offered by the extension;

- **UserAchievements**. Stores a list of pairs *idAch, username* where each row identifies the achievements unlocked by a specific user (i.e. a row *2, Giacomo* means that the user *Giacomo* has unlocked the achievement with *idAch = 2* in the **Achievements** table;

- **UserAvatars**. This table follows the same exact logic as **UserAchievements**, keeping track of the different avatars unlocked by each user;

- **Users**. A table used for keeping track of the different users who signed in to the extension. Associates to each user the currently selected avatar, which is shown to other users;

- **WidgetCrops**. Contains all the information needed for the generation of the scripts used for replaying a past session. Such information includes the dataURL used to generate the screenshot of a widget, the type of interacted widget, the username of the user that interacted with the widget, optional information such as selection index or text content of a field, *CSS* selector and *xpath*. All information about widgets interacted with by a user during a session is deleted when said user ends the current session, in order to keep track of information about current sessions only.

Node.js Server

In order to have data easily accessible to the extension, a local server is run using *Node.js*, an open-source, multi-platform runtime system used to run asynchronous *JavaScript* code. The server is split into two different modules:

- **DAO**, a module which includes two *JavaScript* files that perform queries on the database with the goal of inserting, updating, retrieving and deleting data as users interact with the extension;

- **server**, whose functionality is to act as a bridge between the *DAO* module and the extension, providing the URLs that are fetched by the extension and calling the adequate method from a *DAO* file as a consequence.

```
1     app.get("/api/pages/issues/:username", (req, res) => { //
      called when the extension makes a fetch request with an URL
      that matches the listed one
2     if (utilities.resolveExpressValidator(validationResult(req),
      res)) { //checks if parameters passed are in correct format
3         pageDao.getPageIssues() //call to function that performs
      the query
4             .then((issues) => res.json(issues)) //returns queried
       data
5             .catch((err) => utilities.resolveErrors(err, res)) //
      error handling
6     }
7 })
8
```

Figure 3.2 represents how the server is structured by showing how the three main components (database, DAOs and server) are connected

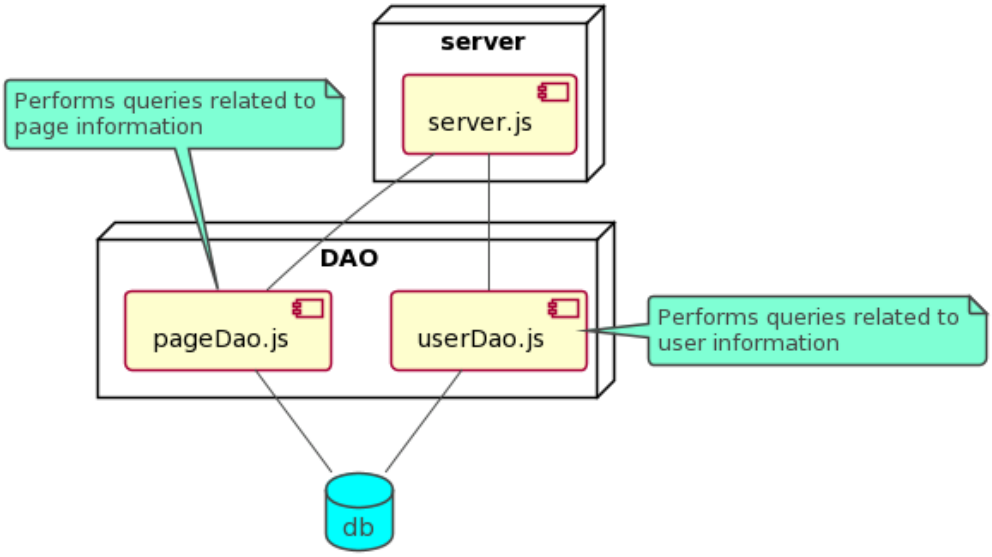


Figure 3.2: Diagram depicting the server structure

3.2 Adopted Gamification Mechanics

The selection of which mechanics to adapt has been based on the idea that users would be more invested in a boring and unfun activity such as web testing if they could perceive a tangible result as a consequence of their actions; another relevant motivation behind the selection of some mechanics was the appeal of competition and desire to prevail.

3.2.1 Avatars

The main gamification mechanic adopted by the extension is the use of personal avatars, which can be unlocked by obtaining achievements and performing relevant tasks for the first time while using the extension. After having unlocked a new avatar, a user can view the entire list of unlocked avatars while in his/her profile page and decide which one to show as current avatar, viewable by other users in order to increase competition and also give a way to increase satisfaction, as having unlocked a rare avatar which can be showed off can be viewed a source of accomplishment for users that put a lot of effort in using the extension. The extension has all users start up with three default avatars, one of which has to be selected as active while creating a user profile, as shown in figure 3.3. All avatars are made starting from the same base image, downloaded from a website that provides free assets to be used for making videogames [14], and then edited through software in order to create small variations of the default avatar; an example of

Figure 3.3: Default avatars selectable while creating a new user profile

such different avatars can be seen in figure 3.4. Users are then able to unlock

Figure 3.4: Examples of unlockable avatars

these different avatars by performing various tasks while testing: an avatar may be unlocked by interacting with at least three different types of widgets during a single session, by declaring a previously reported issue as *solved* or by obtaining an adequate number of achievements, for example. A possible extension of the avatar system, which is intended as a possible starting point for an expansion of the current extension, would be a rework that substitutes the monolithic *PNG* files that make up an avatar limited to just a single distinctive feature (i.e. a crown, a wizard hat, a star decoration on its clothes) with a system where accessories are unlocked rather than complete avatars; such accessories would then be able to be combined onto the default avatar, greatly increasing the amount of customization allowed. Such a reworked system could also see the creation of multiple possible avatars, whose looks would greatly differ, with the goal of bringing future users to increase their efforts, in order to have them unlock newer and rarer features to show off.

3.2.2 Achievements

Achievements are one of the most important elements used in gamification, as they provide a tangible way to reward users for their actions, and are thus used in the extension in a relevant way. The achievement system adopted by the extension works together with Chrome's *notification* API by sending the user a system notification after a significant event that has happened for the first time and is thus worthy of being celebrated with an achievement. Achievements are also graded by a rarity level where the harder to obtain ones (i.e. obtaining perfect coverage on a page or a type of widget inside a page) are identified by a golden medal, while easier achievements have a silver or bronze one; they are then shown, together with the criteria for having unlocked them, in a section inside the extension's homepage dedicated to viewing the user's profile, shown in figure 3.5.

3.2.3 Progress Bars

Another relevant mechanic adopted was selected after deciding that users would need a way to easily see and understand their progress in terms of interactions inside a page. Such progress is divided into two different sections:

- A global progress bar, present in each page tested by the extension, which shows the percentage of widgets interacted with by the user in relation to the total amount of widgets present in the page, with figure 3.6 showing an example of how the progress bar looks in a tested page;
- A set of progress bars located in the extension menu, one for each different kind of widget registered by the extension, detailing the percentage of interaction

Figure 3.5: Homepage section dedicated to listing user achievements

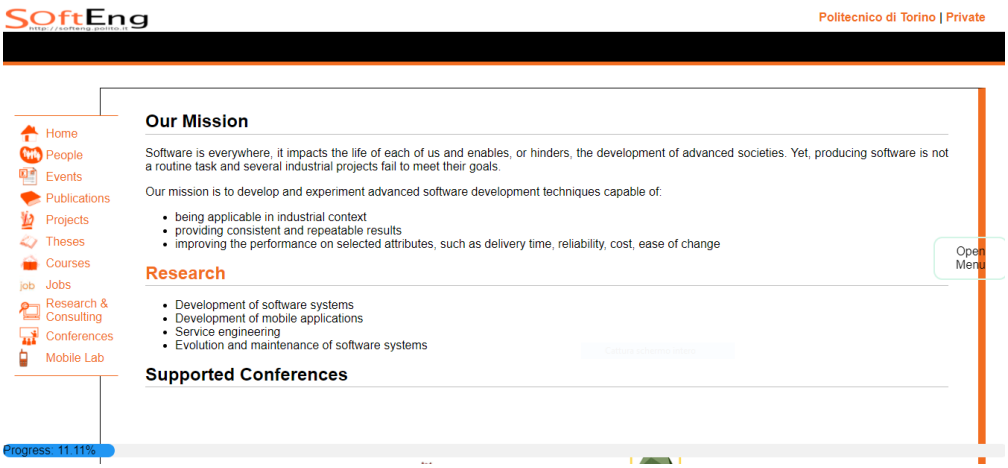


Figure 3.6: Progress bar showing the percentage of interacted widgets in a page

with each specific type for the current page. Figure 3.7 shows an example of these progress bars, along with the message shown when a page does not have a type of widget.

Whenever the user interacts with a new widget for the first time both the global progress bar and the progress bar related to the type of interacted widget are updated straight away, in order to give a sense of actual progress and fulfillment. A limitation of these progress bars, however, is that they only compute the coverage and give no actual information about which elements have actually been interacted with in the page, meaning that their functionality is meant to be a simple indication of progress done, rather than a guide of what remains untested in a page.

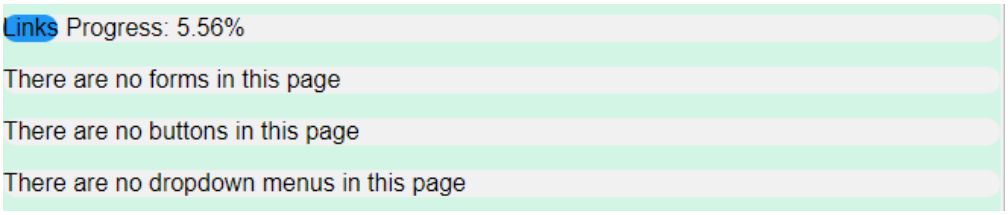


Figure 3.7: Progress bars showing the percentage of interaction for each kind of widgets present in a page

3.2.4 Scores

Scores are intended as the count of widgets found in a page during a session and the count of pages visited in a session. Both scores are also divided into new ones found during the current session, the total amount found during the current session including both new ones and previously found ones and the total amount found during all sessions; scores for both widgets and pages are reported into tables that can be easily viewed while in the extension menu, as seen in figures 3.8 and 3.9.

Page Widgets	Current Session	New	Total
Links	1	0	2
Forms	0	0	0
Buttons	0	0	0
Dropdown Menus	1	0	1

Figure 3.8: Table showing the scores for widgets interacted with in the current page

Once the user ends a testing session the extension shows a recap screen detailing, for all kinds of widgets, the count of total interactions with widgets made in all pages and the total count of new widgets found during the session; a similar count is also present in order to list the count of visited pages and of new pages, as can be seen in figure 3.10. The extension also keeps track of the highest scores obtained by a user during the entire history of his/her testing experience, and these records are shown in a dedicated section in the extension's menu, as seen in figure 3.11.

3.2.5 Leaderboards

One of the main elements used in gamification, leaderboards also make up a relevant part of the extension, as they were selected in order to increase competition by giving users a way to compare their standing and their scores with other users of the extension. There are three different leaderboards, visible inside a dedicated section of the extension homepage, and they rank users based on three different

Pages Visited	Current Session	New	Total
Pages	7	4	53

Figure 3.9: Table showing the scores for pages visited in the current session

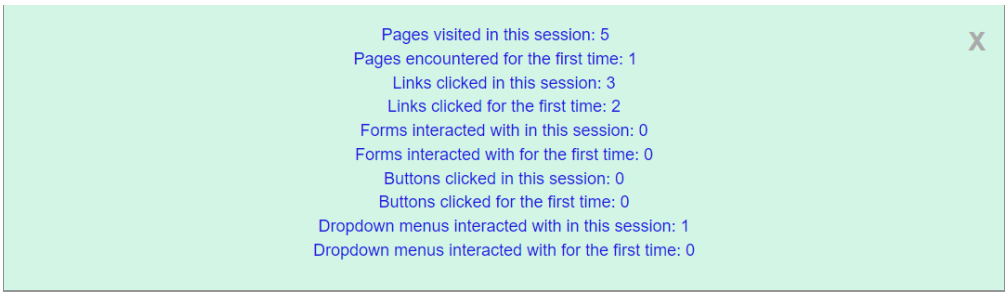


Figure 3.10: Recap screen detailing scores obtained during a testing session

metrics:

- highest amount of new pages found during a session;
- highest amount of new widgets interacted with during a session;
- highest page coverage obtained.

All of the three leaderboards show, as can be seen in figure 3.12, the username, the score and also the current avatar selected by each user: such a decision was made in order to both allow users that have made good progress and unlocked different and rare avatars to show off the fruits of their endeavors and also to increase the feeling of competition in users that have done fewer activities and have unlocked less rare avatars. Another decision taken in order to increase competition between users was to have, when a user's rankings are updated at the end of a testing session, a check on the entire set of rankings, followed by rewarding the user with a new rare avatar connected to the new position: more in detail, placing on one of the top three positions in a leaderboard for the first time will reward the user with an avatar possessing a golden, silver or bronze medal, depending on the position obtained.

3.2.6 Page Discovery

The final gamification mechanic implemented by the extension consists in a special way to signal, when a user reaches a page, whether that page has never been visited before or not by the user during past sessions: a small star, visible in figure 3.13, appears in the bottom-right part of the web page if it is a new one. This star persists as long as the user remains on the web page, as it serves the purpose of signaling whether a page has been visited for the first time or not, a statistic that is

Figure 3.11: Menu section showing the current user's records

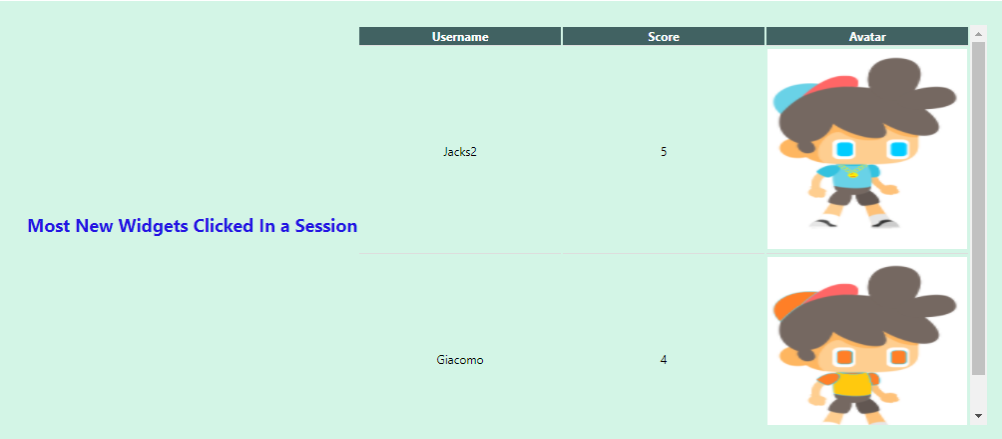


Figure 3.12: Leaderboard of all users sorted by amount of highest new widgets found

kept track of during a session and is used to place users in leaderboards, as already said in previous sections.

3.3 Extension Architecture

The relevant components that make up a Chrome extension have already been listed back in section 3.1.1, what remains unexplained is both the structure of the different components in terms of both interaction between them and functionalities performed by each of them; a representation of how all components are interconnected is depicted in figure 3.14. Section 3.4 will explain in detail how each component works and which specific functionality it implements, while this section will report the content of each node shown in figure 3.14.

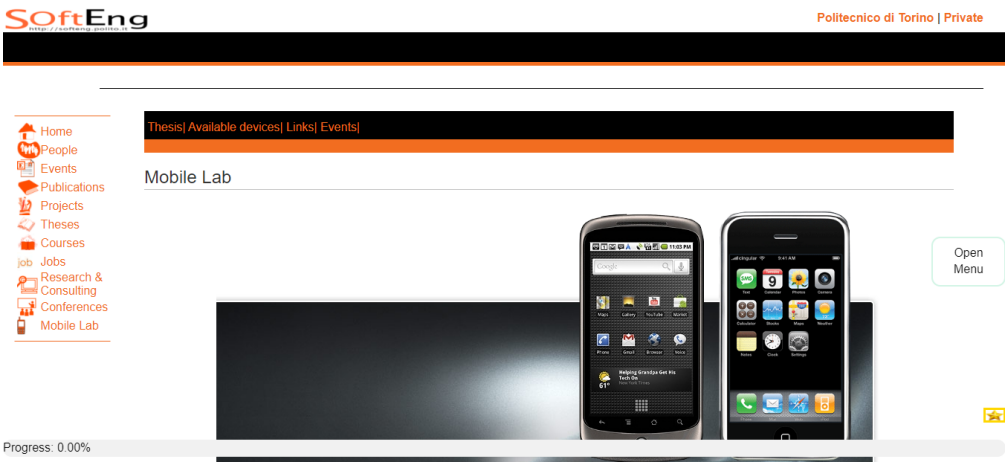


Figure 3.13: Web page with the star signaling a page visited for the first time

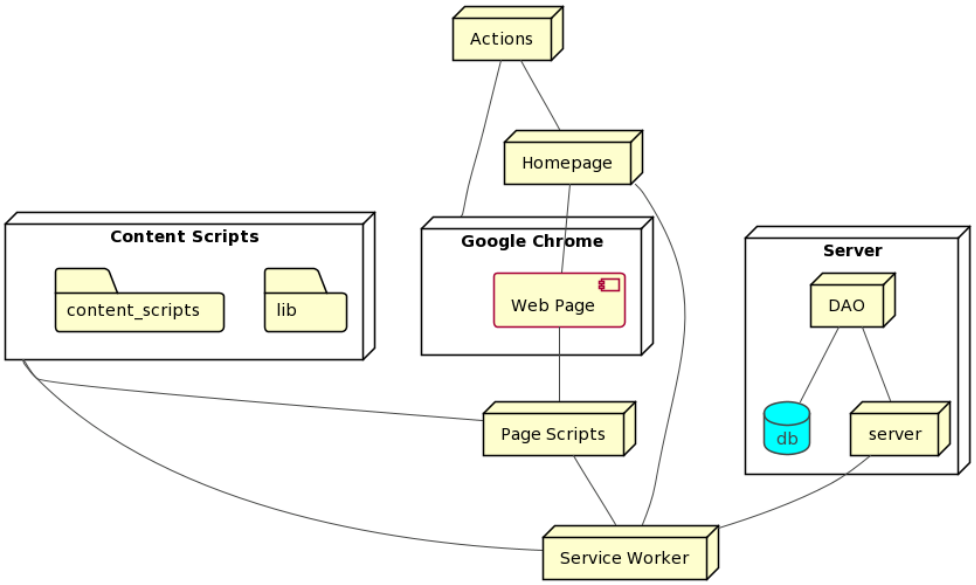


Figure 3.14: Diagram representing the different components of the extension and their relations

3.3.1 Action

The way the extension implements its action is through a simple popup button embedded into Google Chrome's action bar which, when clicked, redirects the user to a new browser tab containing the homepage of the extension.

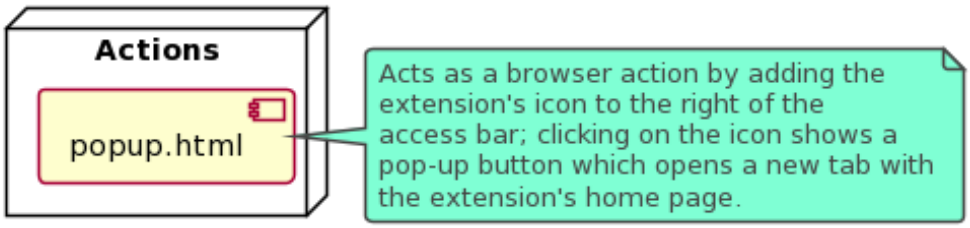


Figure 3.15: Action used by the extension

3.3.2 Homepage

A local web page made up of standard web language techniques, sends requests to the Service Worker in order to retrieve user data and redirects to the web page that is to be tested.

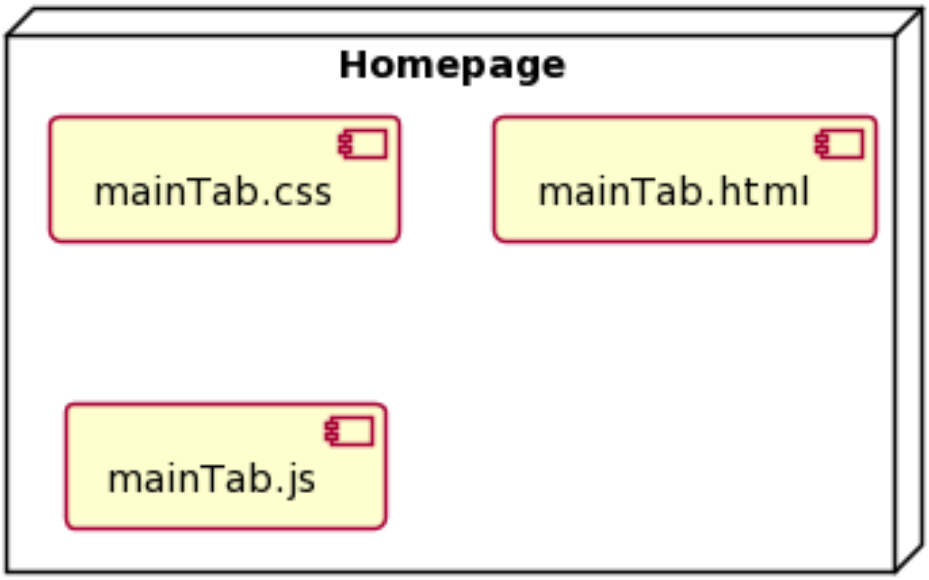


Figure 3.16: Files that make up the extension's homepage

3.3.3 Content Scripts

As was previously mentioned back in section 3.1.1, content scripts make up a fundamental part of a Chrome extension, as they contain functions and variables that are available for the entire context of the extension, with figure 3.17 showing which are the content scripts used by the extension and their structure. What the figure shows is a division into two folders:

- *lib*, a folder containing external libraries used to download a folder compressed in *.zip* format[15][16]. Such functionality is used at the end of a testing session in order to download a script that can be used with SikuliX;
- *content_scripts*, which contains a file tasked with defining the listener to be attached to each interactable element in a web page (i.e. links, buttons, forms) to react to events such as clicks or submit operation, and a second file whose role is to define functions that are used multiple times during a testing session and thus have to be shared and easily accessible.

These content scripts are declared as runnable on all possible URLs, by using the declaration:

```
1 "matches": ["*://*/*"],
```

seeing as the extension is supposed to run on all kinds of websites; this, however, forces a distinction between these shared functions and the functions that perform changes to the tested web page: the files that implement the logic behind these changes are called *Page Scripts*.

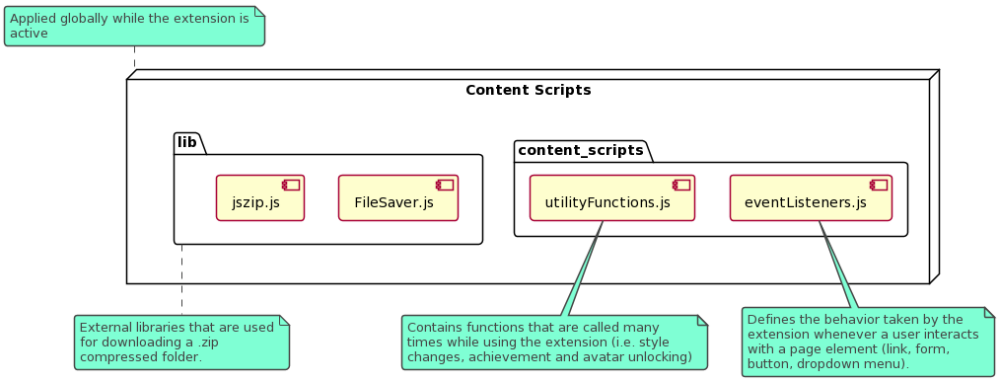


Figure 3.17: Content scripts used by the extension

3.3.4 Page Scripts

Page scripts are peculiar files that act in a similar way to how content scripts work, with the main difference being the fact that, while content scripts are always available to all web pages while the extension is active, page scripts are launched by the service worker if and only if the active tab's URL belongs to the same domain as the one specified by the user at the beginning of a testing session (for example, if the user specifies *https://softeng.polito.it* as starting URL, page scripts

will be launched when the user visits the URL *https://softeng.polito/theses.html*). A graphical representation of which are these page scripts with a short description of their functionalities is depicted in figure 3.18. Another difference between page scripts and content scripts lies in the fact that the latter ones' contexts are available to all other scripts, both page and content, running inside an extension, while page scripts, by virtue of being launched by Chrome's *scripting* API, which causes each page script to have an enclosed context visible only by itself. This means that a variable or function declared inside a page script is visible only inside said page script, which allows for a division of duties among the different scripts.

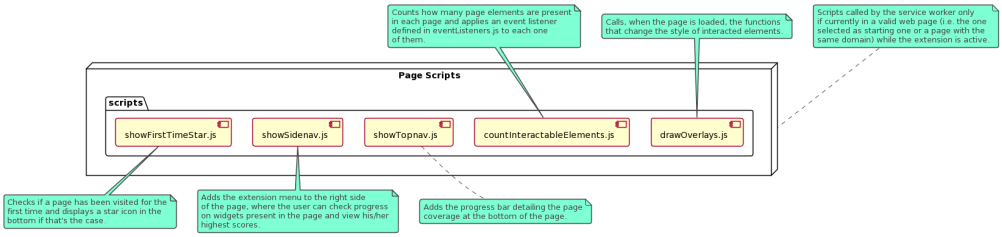


Figure 3.18: Scripts called in a page currently being tested

3.3.5 Service Worker

The service worker can be seen as the most important part of the extension's architecture, as it is a perpetually running file that listens to events that happen during the user's browsing experience. The service worker performs two relevant operations:

- Every time the current tab changes or is updated, it checks if the current URL belongs to the same domain as the one of the URL selected as the starting one and, if such condition is verified, launches the page scripts;
- It listens to messages sent by page scripts through Chrome's *runtime* API in order to send notifications, download files, capture screenshots of the currently active tab or fetch data from the external server.

3.4 Extension Functionalities

Many of the functionalities offered by the extension have been presented in Section 3.2, but other generic augmented testing functionalities are also offered by the extension. What follows is an explanation of the remaining functionalities.

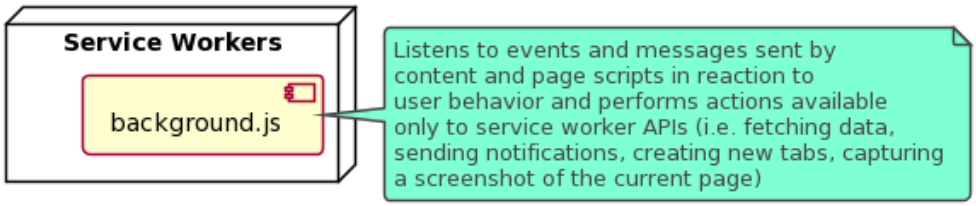


Figure 3.19: Service worker used by the extension

3.4.1 Interaction Overlays

The extension offers a functionality that marks which widgets have already been interacted with by the user: more in detail, all widgets have their *style* property modified in order to have an additional border around them; such border specifies that the element has been interacted with and has a different color depending on the type of widget, with the mapping reported in table 3.1. The extension

Widget Type	Border Color
Link	Red
Form	Green
Button	Blue
Dropdown Menu	Yellow

Table 3.1: Mapping between widget type and border color

menu offers three buttons that allow activating these overlays, with three different possible modes:

- **None**. The default page behavior and styling. In this mode there are no changes done at all to widgets, meaning that there is no distinction between interacted and non-interacted widgets;
- **Interact**. This mode applies overlays to all the widgets that the user has interacted with, and only those, making it easy to know which widgets the user has interacted with in a given page and which ones are missing; an example of how a page looks like while in this mode can be seen in figure 3.20
- **All**. This mode applies overlays to all widgets present in a page, regardless of actual user interaction. Useful in order to highlight all elements in a page the user can interact with.

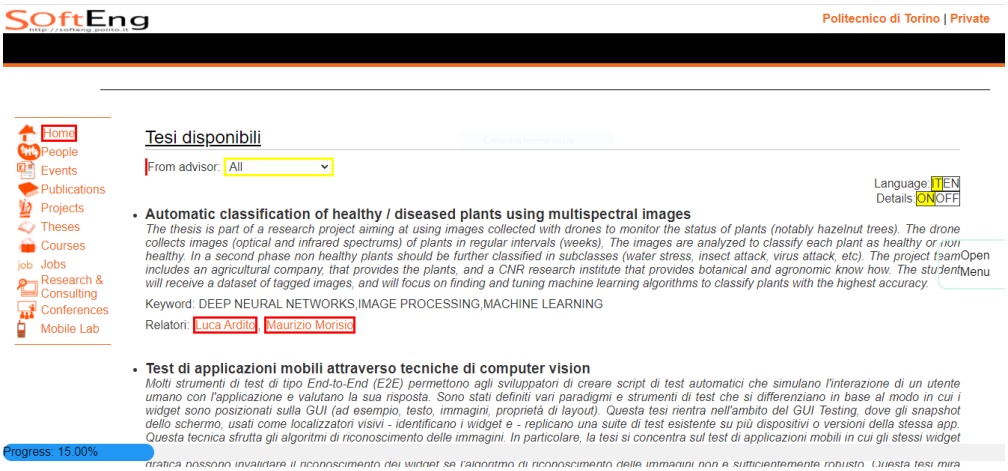


Figure 3.20: Web page with borders marking interacted elements

If the interaction with a widget is performed while in **Interact** mode the border is applied to the element after a delay of three seconds and, also, if the widget was already interacted with in the past and already has a border applied such border is removed before taking the screenshot and then reapplied; such measures are taken in order to ensure that the screenshot of the widget is taken while the latter is in its natural state, so that SikuliX script are able to run without issues on the page. Once selected the mode remains set until it is changed again, meaning that it persists after changing the web page through a link during a session or even after ending a session and starting a new one. Figure 3.21 shows the change in style of a dropdown menu after selecting one possible index.

Figure 3.21: Dropdown menu changing style after user interaction

3.4.2 Issue Signaling

Another relevant feature of the extension, which gives more focus to the concept of testing rather than being based on gamification, is the existence of a second mode of interaction with the web page where the user can report issues found during a testing session. It may happen, for example, that the user is redirected to a wrong page after clicking on a link, or that a button expected to have some effect

on the page content does not perform as intended, or many other cases of issues that web testing aims to find; in this case, the second mode is expected to be used in order to mark a widget as not working so that other users working with the extension may see the issue. Issue reporting works as shown in figure 3.22: the user clicks on the widget found not working correctly and a modal is opened asking for input explaining what the issue actually is; after the issue is submitted the modal is closed and the style of the signaled element is changed in order to have a colored background that, while in reporting mode, signifies that the element has an issue and is visible by other users of the extension; additionally, hovering with the mouse over the widget shows a tooltip presenting the issue. Issues opened and added to widgets can be seen even if they are reported by a user different than the current one, with the catch that an opened issue can be marked as **Solved** only by the same user that reported it; a small limitation of the issue system is that a widget can have at most one issue attached to itself, even counting multiple users, meaning that if a user reports an issue for a link in a page, for example, another user will neither be able to report another issue for said element nor be able to declare its resolution.

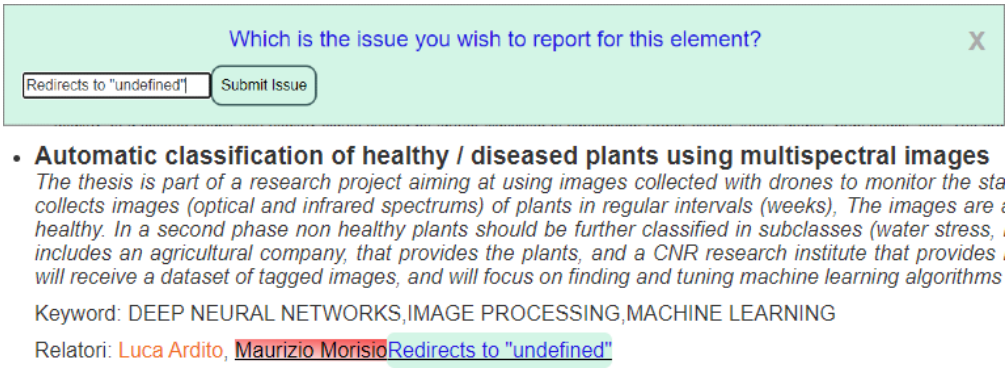


Figure 3.22: Procedure of reporting an issue on a link

3.4.3 SikuliX Script Generation

One relevant feature of the extension is the ability to generate scripts compatible with the visual testing programs SikuliX and Selenium in order to have a representation of a testing session that can easily be replayed in an automated way. The generation of a SikuliX script is done at the end of a session and consists of the creation of a *Python* file that follows the SikuliX constraints and rules for replaying a session; an example of such file is reported below:

```
popup("Beginning replay of past session")
max = 20
wait(5)
while (max > 0)
    if(not exists(Pattern("img1.png").similar(0.55), 0):
        wheel(WHEEL_DOWN, 1)
        max -= 1
    else
        max = -1
        break
click(Pattern("img1.png").similar(0.55), 0)
wheel(WHEEL_UP, 20-max)
max = 20
wait(5)
while(max > 0)
    if(not exists(Pattern("img2.png").similar(0.55), 0):
        wheel(WHEEL_DOWN, 1)
        max -= 1
    else
        max = -1
        break
type(Pattern("img2.png").similar(0.55), "Torino" + Key.ENTER)
wheel(WHEEL_UP, 20-max)
popup("Ending replay of past session")
```

The script emulates a session where the user first clicks on a widget identified by a screen named *img1.png* and then types a keyword in a text field belonging to a form identified by the screen *img2.png* in order to submit the content of a search bar. Script generation adopts some techniques in order to ensure that a script can run to the end and emulate as precisely as possible the actions performed by a user; more in detail:

- **wait(5)**. This command is used in order to ensure that the script waits 5 seconds before executing any action so that, for example after a page change, the script does not start looking for an image before it is actually present on the page and returns an error;
- **while(max > 0)**. A conditional block repeated for each widget whose role is looking for a match with the screenshot of the widget and the page content; if there is no match the condition

```
    not exists(Pattern("img2.png").similar(0.55), 0
```

is verified and the script scrolls down in order to find another match, assuming that the widget is in a lower part of the web page and thus not currently

visible. This sequence of checking and eventually scrolling down is repeated until a match is found, meaning that the script can then execute the action on the identified widget;

- **wheel(WHEEL_UP, 20-max)**. An action taken after the interaction with an identified widget is performed, it is made in order to correct the scrolling down made in order to find the previous widget so that the research of the following one can be done from the beginning of the page. This solution has been implemented as it is not possible to know, while interacting with a web page, the absolute position of a widget in the page but just the position relative to the part of the page that is currently visible, meaning that it is not possible to tell SikuliX if a widget is above or below the one interacted with before.

Issues

Being a program based on image recognition where images are assumed to be captured without interaction, SikuliX struggles a bit in some cases since a screenshot of a widget is taken when there is interaction with it; this causes the following issues:

- clicking on a link causes it, generally, to change color, depending on the *CSS* rules defined by the developer of the tested web page; for example, a page may define the default colors of a link as

```
1     /* unvisited link */
2     a:link {
3         color: red;
4     }
5     /* mouse over link */
6     a:hover {
7         color: blue;
8     }
9     /* selected link */
10    a:active {
11        color: blue;
12    }
13
```

This definition means that the screenshot of a link, taken when the link is clicked, has a blue colored link while said link has normally a red color, meaning that when the SikuliX script is run finding a match may not be possible, as the different colors can require a lower similarity value in order to have a correct match, with the added risk of matching with other parts of

the page. In order to work around this issue and have correct matches with links the extension does not just take a screenshot of the *HTML* document element that identifies a clicked link but doubles the height and width of the screenshot and changes offsets so that the link remains in the center of the new, larger screenshot and matches are easier to find;

- writing text into a text field and then deciding to change said text can cause an issue. An example of what causes the issue is explained as follows:

  1. the text field is clicked, the extension takes a screenshot of the empty text field;
  2. the text field is filled, the extension stores information about the text content associated with the text field;
  3. the text field is clicked again with a new screenshot being taken;
  4. the new value of the text field is registered and stored.

  In case SikuliX has to replay such a sequence of operations it may not find a match as it handles the sequence in this way:

```
1    type(Pattern("img1.png").similar(0.55), "a", KeyModifier.CTRL
     )\n
2    type(Pattern("img1.png").similar(0.55), Key.BACKSPACE)\n
3
```

  The first **type** command emulates selecting all the content of the text field as if typing *CTRL + A* while the second command cancels all highlighted text; the issue comes up when performing the second **type** command, as there is no screenshot of the text field with all text selected and highlighted, meaning that finding a match is not guaranteed;

- Using a *wait(5)* as a command performed after every operation may not be enough, due to the fact that different devices may perform SikuliX scripts at different speeds and connectivity issues may also affect page loading, causing the script to wait too little and then fail because the page was not loaded correctly.

3.4.4 Selenium

The extension also generates a Selenium script at the end of a testing session, in order to have a second way to replay past sessions. The generation of a script is made by creating a *.side* file with a format compatible with what the Selenium

IDE requires by creating an array of commands, one for each interaction with a widget, with the following format:

```
"commands":[
    {
        "id": "idCommand0",
        "comment": "",
        "command": "open",
        "target": "http://softeng.polito.it/",
        "targets": [],
        "value": ""
    },{
            "id":"idCommand1",
            "comment":"",
            "command":"click",
            "target":"css=#sidebar > TABLE:nth-child(1) >
    ↪ TBODY:nth-child(1) > TR:nth-child(6) >
    ↪ TD:nth-child(2) > A:nth-child(1)",
            "targets":[
                [
                    "css=#sidebar > TABLE:nth-child(1) >
    ↪ TBODY:nth-child(1) > TR:nth-child(6) >
    ↪ TD:nth-child(2) > A:nth-child(1)",
                    "css:finder"
                ],[
                    "linkText=Theses",
                    "linkText"
                ],[
                "xpath=/html/body/table[3]/tbody/tr[1]/td/
                span/table/tbody/tr[6]/td[2]/a",
                    "xpath:idRelative"
                ]
            ],
            "value":""
            }
]
```

The first command of the array is always a command that opens the web page selected by the user as the starting page of a testing session, while following commands depend on the action performed and the type of widget; more in detail:

- Links and buttons are simply clicked, with no extra action being done to them, through the *click* command;

- Form fields are handled differently depending on whether they are checkboxes/radios or other types; the former simply handle the action by using the *click* command on the identified element while other types are handled through the *type* command, with the *value* parameter being filled with the content of the field rather than being empty as happens with other cases; finally, in order to handle form submission, a special command is added right after the command corresponding to the final field of the form in order to execute form submission, with the following format:

```
{
    id: `idCommandSendKeys`,
    comment: "",
    command: "sendKeys",
    target: "id=mod-search-searchword104",
    targets: ["id=mod-search-searchword104", "id"],
    value:"${KEY_ENTER}"
}

```

  The command *sendKeys* works by emulating the pressure of the *ENTER* key on the last textual field of the form, causing the submission of the form;

- Dropdown menus make use of the *select* command, whose function is to select one of the options, identified by its label, presented by the dropdown menu.

Identification of a widget depends on the value specified by the *target* parameter of a command object: said parameter can have different values, as long as they are ways to univocally identify an *HTML* document element; the extension employs, where possible, the following values to identify widgets:

- **CSS selector**. Adopted as the default value used in order to identify widgets as it can be computed for every element part of an *HTML* document; it is computed with the following algorithm:

```
function selector(el) {
let names = []
while (el.parentNode) {
    if (el.id) {
        names.unshift('#' + el.id);
        break;
    } else {
        if (el == el.ownerDocument.documentElement) {
            names.unshift(el.tagName)
        } else {
```

```
                let c, e
                for (c = 1, e = el; e.previousElementSibling; e = e.previousElementSibling, c++);
                names.unshift(el.tagName + ":nth-child(" + c + ")")
            }
            el = el.parentNode
        }
    }
    return names.join(" > ")
}

```

The algorithm iterates, starting from the element whose selector is to be computed and going up through the document until it finds an element with an identifier or the root of the document; it also performs, before checking the parent element of the current element, a count of how many siblings the current element has before its position (i.e. the fourth row of a table with 6 rows will be the fourth child of its parent, the table body) in order to correctly identify the position of the element in the document. Each step of the iteration adds a new element to the *names* array through the *unshift* command, which puts the new value at the first position of the array it is applied to and, at the end of the loop, all elements of the array are made into a single string by using the *join* function on the array, which returns a string that is suitable for use by Selenium;

- **xpath**. The generation of an *xpath* starting from an element follows the same logic and reasoning as the one behind a *CSS* selector (starting from the current element and going up until an ancestor with an identifier is found); it is used as an alternative way to identify widgets that is applicable to almost all cases and is performed by the following algorithm:

```
    function xpath(el) {
    if (typeof el == "string") return document.evaluate(el, document, null, 0, null)
    if (!el || el.nodeType != 1) return ''
    if (el.id && el.tagName.toLowerCase() === "div") return "//div[@id='" + el.id + "']"
    let sames = [].filter.call(el.parentNode.children, function (x) { return x.tagName == el.tagName })
    return xpath(el.parentNode) + '/' + el.tagName.toLowerCase() + (sames.length > 1 ? '[' + ([].indexOf.call(sames, el) + 1) + ']' : '')
}

```

The algorithm performs the iteration through recursion until it finds an element with an identifier, at which point it returns, causing a series of return operations that end up with the creation of a string that starts at the element with an identifier found or the document root and goes all the way down to the position of the element whose xpath is needed;

- **identifier**. If the element has an identifier then said identifier will also be used as a possible target for Selenium to adopt in order to find the element;
- **link text**. Generally, links do not have identifiers in *HTML* documents, so they need another way to be easily identifiable by Selenium as an alternative to CSS selectors and xpaths; another possible identifier is the text content of the link itself, as there should not be two links in the same page with the same exact text (and even if there were, it would be logical to assume that they redirect to the same page, so Selenium would still have no issue in finding the link to interact with when emulating the session).

All the values used to give Selenium a way to identify which element to interact with in each step of a session are computed at the same time as when the screenshot is taken for SikuliX, meaning that they are computed just as the user interacts with widgets and are then stored in the external database until the end of the session; when the session ends and all files are downloaded all information related to which widgets have been interacted with and in which order during the session that just ended is deleted from the database so that future session can restart from zero.

3.5 Testing Session

A testing session performed by the extension is registered in terms of interactions done by the user with the widgets present in a page: more specifically, it records, in the order in which they are executed, clicks on links and buttons, selections made on dropdown menus and values inserted in form fields, along with the submission of the form itself. The registration of interactions is then elaborated at the end of a session in order to generate script files that can be used by the visual testing programs introduced before, SikuliX and Selenium IDE. An example of a session, whose aim is to perform navigation on a website, navigate through its pages by clicking on links and interacting with widgets, whenever they are found, is performed in the following way:

1. The user opens up the extension's home page by clicking on the popup button that appears when clicking on the logo in Google Chrome's action bar;
2. The user pastes in the text field the URL of the starting page to test and presses the button that will change tab to one having said URL;

3. Once on the page the user can interact with the different widgets present there, with the extension acting in different ways according to the type of widget. More in detail, the extension registers, for all types, the index of the clicked widget related to the total amount of widgets present in the page for that type, the *CSS* selector, the *xpath* required for the creation of a *Selenium* script and also takes a screenshot of the visible web page; said screenshot is resized to contain only the specific widget so that *SikuliX* can properly know where to act. In addition to this, there are specific actions taken based on the different types:

    - **link**: the redirection to a new page is delayed for 0.5 seconds before taking the screenshot;
    - **form**: the extension keeps track of the value inserted as content of the form and also, when the form is submitted, it performs a check on whether the form was submitted with a button click (registered as a normal button) or with the use of the *enter* key, in which case the extension marks the last field of the form as the one where the key will have to be submitted from;
    - **button**: the click is registered with no additional action made by the extension;
    - **dropdown menu**: the extension registers the index selected by the user, to perform the same action with SikuliX and Selenium.

4. Each interaction with a widget can be performed as many times as the user prefers, whether in the starting page or in one reached through interaction with a widget. The extension, however, does not register actions such as hovering over elements or scrolling up and down, with the scripts handling these behaviors in their own way;

5. During the session the user can, by opening up the menu, check the progress made in the current page and also view his/her records obtained during past sessions;

6. When the user decides that the session is finished all he/she has to do is to open up the side menu and end the session by clicking on the *End Session* button; the extension will end the session by showing a modal containing a recap of all interactions made in the session (widgets clicked and pages visited, including a count of all the new ones). Along with showing the modal, the extension also downloads a *.zip* folder containing the screenshots of all widgets clicked and a *Python* script to be used by SikuliX and a *.side* script compatible with the Selenium IDE.

Figure 3.23 represents in detail how a testing session works: the **Page** block is repeated as long as the user remains in a page and consists in the user deciding, as long as he/she still wants to interact with widgets in the page, to keep the session going. The second example of session is made by using the feature of the extension

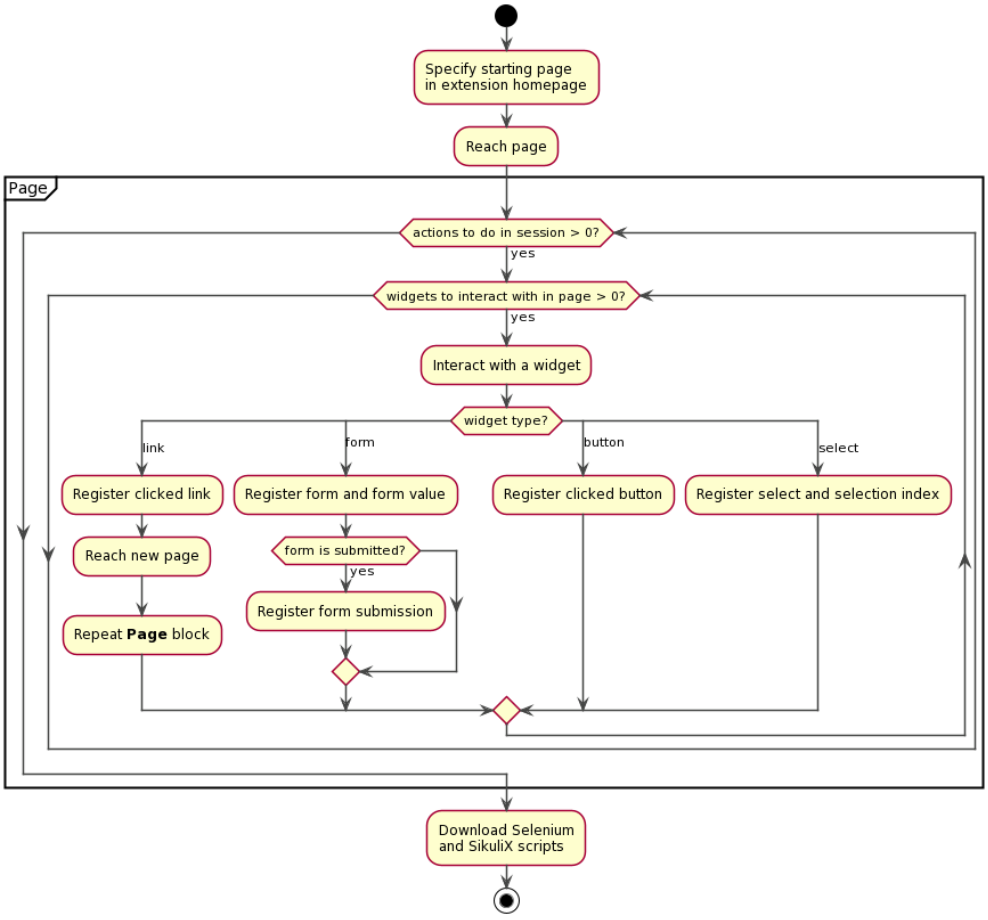


Figure 3.23: Activity diagram representing a normal testing session

that allows a user to report an issue connected to a widget; in this case, the session goes as follows:

1. The user opens up the extension's home page by clicking on the popup button that appears when clicking on the logo in Google Chrome's action bar;

2. The user pastes in the text field the URL of the starting page to test and presses the button that will change tab to one having said URL;

3. The difference between the previously explained session is as follows: while

interacting with a widget the user can find that it does not have the expected behavior and thus has to be reported as not working as intended;

4. Once the user has found a widget not working correctly he/she can report the issue by changing the mode of the extension to **Signal Issue**: in this mode, clicking on a widget does not have it execute its expected behavior but opens up a modal where the user can write the issue found with the widget;

5. The user can also, while in this mode, mark issues he/she had reported in the past as **Solved**, by clicking on widgets that were previously reported as having an issue;

6. Just as was the fact in the previous example, the user can terminate a session in the same way, with the extension downloading the scripts created based on the actions made while in **Interaction** mode, as clicks and behaviors made in **Signal** mode are not registered as actions to be emulated in such scripts.

An example of how the session just explained works can be seen in figure 3.24, where **Default behavior** refers to how the extension acts, as seen in figure 3.23, by registering clicks and actions.

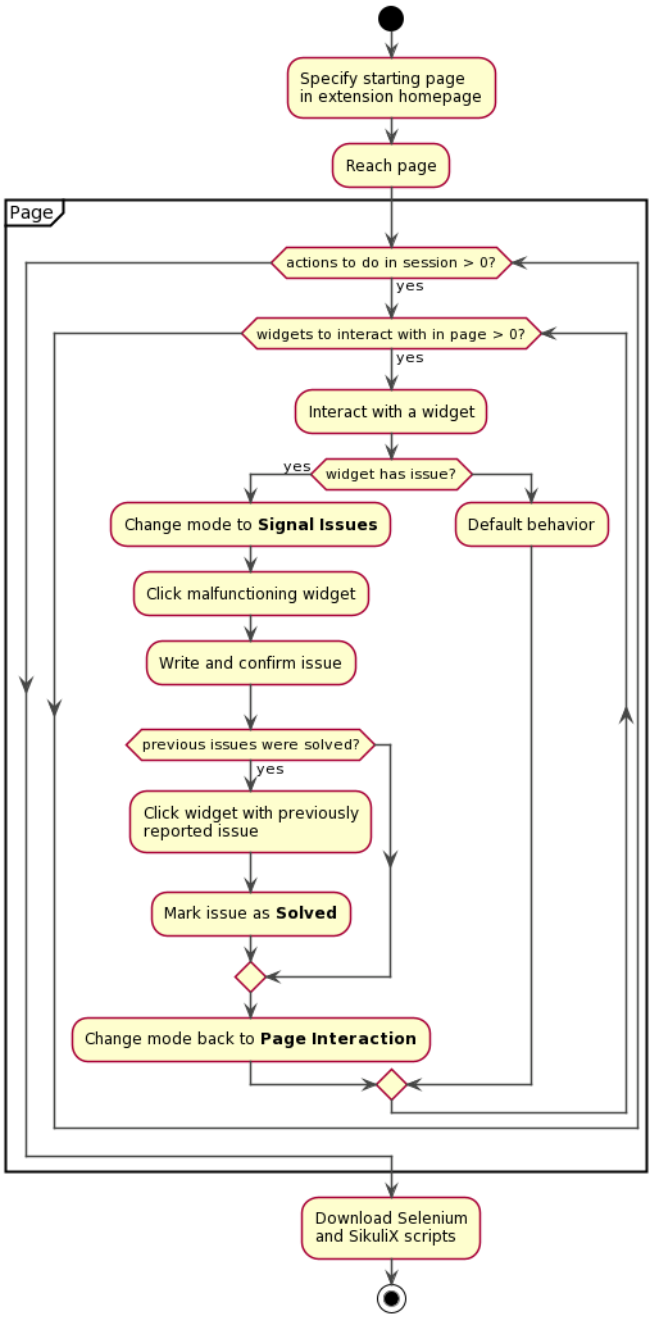


Figure 3.24: Activity diagram representing a testing session where issues can be reported and solved

Chapter 4

Evaluation

In order to assess whether the extension meets good enough standards in terms of usability, as well as to see how gamification mechanics are actually received by users and what are the benefits, if any, they bring, a preliminary usability evaluation has been conducted. The goal of such evaluation was to gather feedback on how users perform when using the extension and what could be improved upon with future development plans.

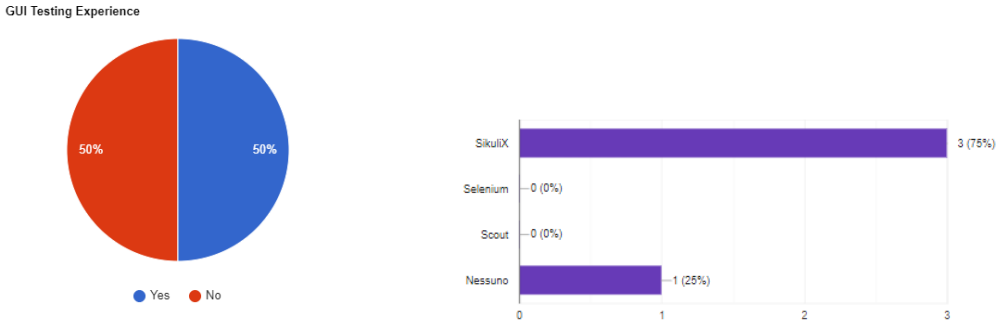
4.1 Participants

Since the extension is developed for GUI testing it was decided that evaluators had to be searched for among developers or generally people that would be close to actual future users of the extension, being knowledgeable of the field. Selected subjects were students enrolled in a Master's degree in Computer Engineering, one graduate of said Degree already working as a software developer and one student in Game Design, whose input on the gamification elements was judged as particularly relevant in order to understand if there was room for improvement in their implementation. The graduating students also followed different orientations:

- One student followed the Software Engineering course;
- Two students followed the Cybersecurity course; additionally, one of them was working part-time as a software developer and the other as an application developer focused on Cybersecurity systems;
- One student followed the Data Analytics course and was already working part-time as a software developer.

Of these participants, only three of them had experience in terms of GUI testing, having followed a course on Software Engineering which presented the testing

approach, from basic unit testing to complete GUI testing of applications, as reported in figure 4.1. All the participants that also had working experience added that GUI testing was not performed in their line of work. It is also worth mentioning, however, that all the participants with a Computer Engineering background also had knowledge on web development, with two of them mentioning the usefulness of a practical tool for easily generating scripts, remembering how unfeasible and cumbersome it was to generate SikuliX scripts, citing that scripts were *hard to automate* and that *screenshot and command acquisition was a bother* as examples of critiques. Many participants, including also one that had no past experience on GUI testing, also added that a tool that aimed to perform GUI testing would be greatly improved if it was *easy to use*.



(a) Past experience with GUI Testing **(b)** GUI Testing tool knowledge

Figure 4.1: Experience on GUI testing and knowledge of tools

4.2 Evaluation Script

Evaluation sessions were intended to assess two main concepts: the general perceived usability of the system and the reception of gamification mechanics: the latter point was deemed as particularly important to evaluate, as a way to measure the effectiveness had by the mechanics in incentivizing evaluators to test the different behaviors of web pages. Evaluation sessions were all conducted by following a detailed script, which is reported as follows:

1. **Introduction**. A brief explanation of the purpose of the extension, the reasoning behind some choices and what GUI testing is about;

2. **Setup**. A guided setup of the extension on the evaluator's PC: installation of the *Node.js* local server, loading of the extension onto the Google Chrome browser through enabling developer mode;

3. **Session Presentation**. After the extension has started correctly the evaluator will create an account on the extension and access the homepage, where he/she will be able to view the (still empty) list of achievements and the leaderboards with other users. Said operation will be followed by an explanation of how a testing session is structured:

   - The link of the starting page to test is to be pasted in the adequate text field in the extension homepage; clicking on the button close to the field will redirect to said page;
   - The evaluator will be able to interact freely with all the widgets present in the page and encouraged to interact with as many kinds as possible in order to perform different actions while keeping in mind that the extension possesses two limitations: actions are recorded only on pages belonging to the same domain of the starting one and page changes depending not on widgets but browser actions (forward/back button, manual URL changes) are not detected. These limitations do not mean that the extension does not work, however, but that the generation of scripts does not keep track of such events;
   - The evaluator can also find issues with the page and decide, if no other user has already done so, to report them by using the appropriate function;
   - The evaluator can decide to end a session at any time he/she prefers, as long as he/she feels that enough content on the website has been explored, or that a significant enough milestone (i.e. reaching 100% coverage on one page, obtaining a good amount of achievements) has been reached;
   - After the end of a session the evaluator is encouraged to check back on the homepage, where he/she can check whether his/her position in one or more leaderboards has improved and observe avatars and achievements eventually unlocked during the session.

4. **First Session**. The first testing session will be conducted on the

   https://softeng.polito.it[17] website: here the evaluator will be asked to navigate freely to get a feeling of how the extension works and how measures of actions made are taken; this session will go on until one of the conditions for ending are reached;

5. **Second Session**. After the end of the first session and the checking on the evaluator's standings in leaderboards a second testing session will start, this time with the *https://elite.polito.it*[18] website. This other session performed on a website that offers different organization rules and a different structure compared to the previous one is intended to show the evaluator that the extension can perform on different websites with relative ease. During this

session the evaluator will also be encouraged to try and obtain higher scores compared to the past session; hints will also be given in order to guide him/her towards unlocking avatars or achievements that were not obtained in the previous session and towards which an interesting progress level had been obtained;

6. **Script Generation**. After ending both sessions successfully, the evaluator will be asked to check the files downloaded at the end: he/she would then confirm whether the downloaded images used for the SikuliX script matched with the interacted elements on the websites;

7. **Evaluation Questionnaire**. The final step of the evaluation procedure will consist in having the evaluator fill a questionnaire through *Google Forms* where his/her opinions will be gathered in order to gauge the usability of the extension.

4.3 Evaluation Questionnaire

The questionnaire given to evaluators at the end of the sessions aimed to assess the usability of the extension; questions asked included a section dedicated to asking the evaluator about previous experience in GUI testing, a section based around the System Usability Scale, which offers a quick way to measure the perceived usability of a system with a numerical score in the range 0-100 (not to be interpreted as a percentage, however); lastly, the questionnaire includes open-ended questions where the evaluator can list what he/she felt was the most effective part of the extension, eventual problems emerged while using the extension and suggestions about what could be improved or expanded upon. The questions asked in the questionnaire, along with their type, are reported in the following subsections.

4.3.1 GUI Testing Background

1. **Q1.1** Did you already have experience on GUI testing before using this extension? (Yes/No question);

2. **Q1.2** In case you answered *Yes* to the previous question, did you already know about tools used for GUI testing? (Multiple choices question);

3. **Q1.3** In case you already knew about GUI testing tools, would you say they have issues that make them difficult/unpleasant to use? (Open question);

4. **Q1.4** What would a GUI testing tool need in order to convince you to use it? (Open question).

4.3.2 Extension Functionalities

1. **Q2.1** Did being able to unlock new avatars and achievements motivate you to interact more with the visited pages? (Likert scale question);

2. **Q2.2** Did having a progress bar showing the page coverage motivate you to interact with the different elements present in pages? (Likert scale question);

3. **Q2.3** Did having progress bars showing the coverage related to the different types of elements present in a page motivate you to interact more with specific types of elements? (Likert scale question);

4. **Q2.4** Did having a leaderboard in the homepage motivate you to try and obtain higher scores? (Likert scale question);

5. **Q2.5** Did being able to see avatars unlocked by other users motivate you to try and unlock more avatars? (Likert scale question);

6. **Q2.6** Did the star shaped Easter egg shown when visiting a new page motivate you to visit as many pages as possible? (Likert scale question);

7. **Q2.7** Did you appreciate the functionality that marked the elements you had already interacted with in a page? (Likert scale question);

8. **Q2.8** Did having recap tables containing your scores in the side menu motivate you to try and obtain higher scores? (Likert scale question);

9. **Q2.9** Did having a section in the side menu dedicated to showing your records motivate you to try and beat those records? (Likert scale question);

10. **Q2.10** Do you believe that the recap screen at the end of a session had a positive effect in motivating you to try and interact more with pages in future sessions? (Likert scale question);

11. **Q2.11** Do you believe that the automatic generation of GUI testing scripts could be a useful functionality of the extension (even if you never used the tools, SikuliX or Selenium) (Likert scale question);

12. **Q2.12** During the testing sessions, did you happen to find issues in the pages you were testing and report them with the appropriate functionality? (Yes/no question).

4.3.3 System Usability Scale

1. **Q3.1** How much do you agree with the phrase *I think I would like to use this extension frequently*? (Likert scale question);

2. **Q3.2** How much do you agree with the phrase *I think this extension is needlessly complex*? (Likert scale question);

3. **Q3.3** How much do you agree with the phrase *I think this extension was easy to use*? (Likert scale question);

4. **Q3.4** How much do you agree with the phrase *I think I would need the help of an expert in the field in order to be able to use this extension correctly*? (Likert scale question);

5. **Q3.5** How much do you agree with the phrase *I think the various functionalities of this extension are well integrated*? (Likert scale question);

6. **Q3.6** How much do you agree with the phrase *I think there is too much inconsistency in this extension*? (Likert scale question);

7. **Q3.7** How much do you agree with the phrase *I think many people would be able to learn how to use this extension in a short time*? (Likert scale question);

8. **Q3.8** How much do you agree with the phrase *I think this extension is cumbersome to use*? (Likert scale question);

9. **Q3.9** How much do you agree with the phrase *I felt confident in what I was doing while using this extension*? (Likert scale question);

10. **Q3.10** How much do you agree with the phrase *I had to learn many things before being able to use this extension correctly*? (Likert scale question).

4.3.4 Opinion Gathering

1. **Q4.1** What is, in your opinion, the strongest asset of the extension, the thing you feel you appreciated the most? (Open question);

2. **Q4.2** Did you encounter any issues or problems while using the extension? If you did, describe them. (Open question);

3. **Q4.3** Do you have any suggestions on how to improve the extension? If you do, write them. (Open question).

4.4 Results

After conducting the preliminary evaluation with all the participants, results were then examined, with the goal of computing a score based on the System Usability Scale and assessing the actual opinions on the gamification mechanics.

4.4.1 Usability Results

The metric selected to evaluate the usability of the extension is the System Usability Score, defined by Brooke et al[19], a set of ten Likert-scale questions with answers ranging from *Strongly agree* to *Strongly disagree*; even-numbered questions are generally negative in spirit (i.e. question 2, *I found the system unnecessary complex*), while odd-numbered ones are more positive. Computation of the score of a SUS questionnaire is done as follows:

1. Each answer equals to a score, with *Strongly Agree* being equal to 5 and *Strongly Disagree* to 1;
2. For negative questions, subtract 1 from each score;
3. For positive questions, subtract the score from 5;
4. Sum the scores computed from applying the previous points to all the questions, then multiply the result by 2.5.

Since the questionnaire has been asked to multiple participants computing an average score turned out to be not as straightforward as expected, as the sum of points for each question should be a natural number. In order to reach a perceived average SUS score the following measures have been taken:

- For each question the most common answer would be selected as the one used to compute the score;
- In case of situations where there were two equally common answers then the *less extreme* would be selected.

The end result score, computed after taking into account the aforementioned measures, is reported below:

$$(4 + 3 + 3 + 3 + 4 + 3 + 4 + 3 + 3 + 3) \times 2.5 = 82.5$$

Generally, a SUS score of 68 or higher can be considered as an indication of an above-average usability level; it can thus be inferred that the extension has a good enough usability level that it can be easily picked up and used with good result, which is a relevant feature for a gamified system. A more detailed overview

of the answers is shown in figure 4.2: the split division of answers is due to the positioning of questions in such an order that sees each positive question followed by a negative one. Such positioning ensures that evaluators have no bias in answering the questionnaire and can also act as an internal evaluation of the questionnaire. The fact that answers follow the same trend (positive questions are generally agreed upon and negative questions are also disagreed upon) confirms that evaluators did not answer their questions randomly but actually believe that the extension is sufficiently usable.

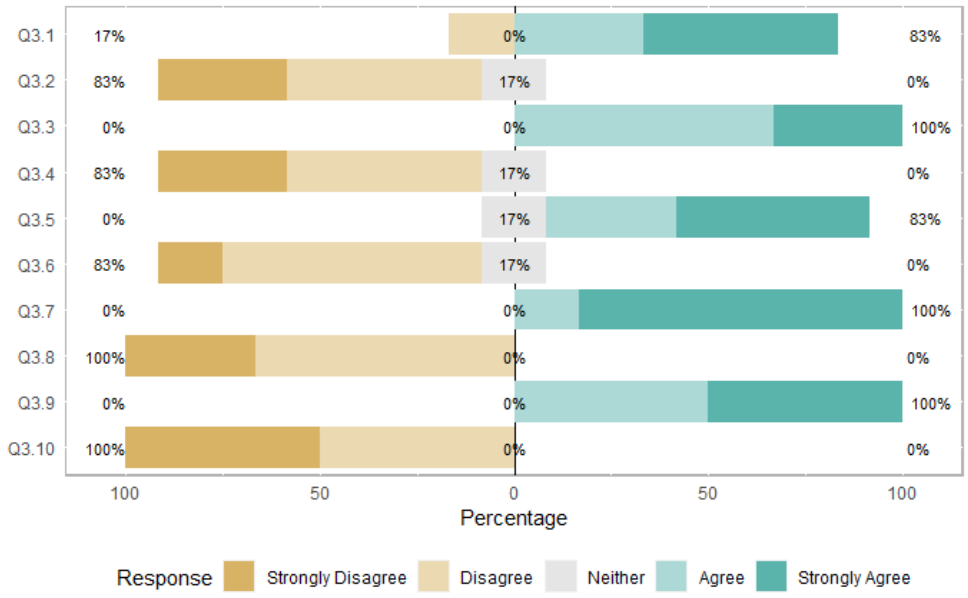


Figure 4.2: SUS Questionnaire Results

4.4.2 Mechanics Results

The second goal of the evaluation, observing the reception of gamified mechanics, has been performed thanks to both the section on the questionnaire related to them and the actual observation of the sessions performed by the evaluators. In terms of actual appreciation, elements such as the leaderboards, the progress bars and unlockable avatars/achievements were the most well received, as can be seen by the results of questions **Q2.1**, **Q2.2**, **Q2.3** and **Q2.4** in figure 4.3. The effectiveness of progress bars and leaderboards was also noted directly during the evaluations: of the six participants, four of them decided to keep going in their first testing session until they had reached 100% coverage in at least one page, motivated by either the global progress bar or by the widget-specific ones, the reasoning for the latter

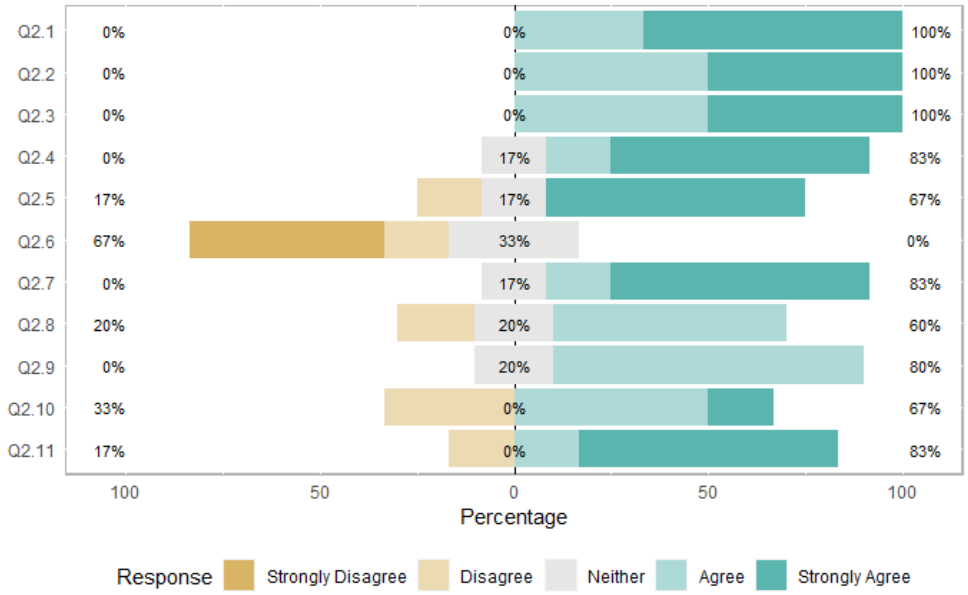


Figure 4.3: Functionality Questionnaire Results

case being that focusing on increasing coverage on the single types of elements is automatically connected to an increase on global page coverage, and it is possible to interact with all forms first, then all links and so on. Regarding leaderboards participants had a positive approach due to competition: some wanted to get to the top just for their own satisfaction, others wanted to at least beat evaluators they knew and had already gotten good scores. On the other hand, however, other mechanics such as the star-shaped Easter egg or the menu section containing records were not met with the same appreciation: evaluators remarked that they paid little attention to the star signaling new pages, which they felt did not have much impact in their experience; the records section was not also much well-received, as evaluators revealed that rather than improving their records they preferred focusing on improving coverage, reasoning that records would improve following their improvements. The non-gamified functionalities features were also generally well received, as shown by answers **Q2.7** and **Q2.11** in figure 4.3. Question **Q2.7** reports that an interesting majority of evaluators appreciated the functionality that highlights widgets, and such appreciation was also noticed during the evaluations: evaluators made intensive use of the functionality, alternating between the setting that highlights all elements that can be interacted with and the one that marks only the ones previously interacted with in order to identify which elements were still missing to reach perfect coverage. An interesting majority of evaluators, as seen in question **Q2.11**, also admitted the importance of script generation: more in

detail, three of the four participants that assigned the maximum value to the script generation feature had already previous experience with SikuliX and remembered how challenging it was to define automated scripts. Said feature was defined by multiple participants as an extremely useful functionality, removing most of the work required for a cumbersome activity such as GUI testing. An interesting result visible in figure 4.2 is that shown by question **Q2.6**, related to the star-shaped easter egg: said mechanic was the least well-received of all the gamified elements implemented in the extension, with evaluators mentioning that it had little impact and it was also hardly noticeable when navigating the various pages.

4.4.3 Considerations

The final section of the questionnaire aimed to understand the general opinions of the evaluators, measuring what they appreciated the most, whether they had encountered some issues during the experiences and what they felt could be improved upon.

Evaluators reported they found the following issues:

- Achievements with a *Silver* rarity level appear as achievements to be unlocked, rather than already obtained;
- Interacting with the extension menu is sometimes counter-intuitive, as well as leaving the menu;
- A reload of the extension homepage is sometimes needed in order to view updated stats and achievements after ending a session;
- It is not possible to know beforehand the criteria for unlocking achievements and avatars, and it is also never mentioned how many achievements/avatars remain to be unlocked;
- After interacting with a link element there is a small delay before reaching the new page.

These issues, together with the following suggestions, will be taken into consideration as possible starting points for future development plans of the extension. The suggestions left by evaluators are as follows:

- Adding a loading icon or an animation during the delay after clicking a link would be a nice improvement, making loading less frustrating and removing the confusion of connection being slow;
- Adding a button in the menu that brings the tester to the page selected as starting one would be a good addition, as it would give an easily reproducible way to leave a page with no way out;

- It would be nice, in the leaderboard relative to page coverage, to know on which page the highest score a user had obtained;
- Another way to unlock avatars, other than the predefined ones, would be to have them be buyable; the currency needed for the purchases could be obtained after unlocking achievements or reaching significative scores;
- It would be interesting if it was possible to share results or leaderboards with friends, with a link or using social media;
- The profile section could be improved if it had a visible list of the goals to reach in order to unlock avatars/achievements; this would increase a user's desire to work with the extension, as he/she would easily know what is still not unlocked and how to unlock it.

4.5 Final Reflections

Overall, what can be gathered from the evaluations and the ending questionnaire is that the more commonly used gamified mechanics (leaderboards, unlockable achievements, progress bars, avatars) have a positive impact on GUI testing, making users more motivated and involved in the testing activity. Such result is in line with what previous studies about gamification have already demonstrated and can be used to justify additional work on the extension, to improve what already exists and refine even more the various features. It must also be remembered, however, that these results come from a simple preliminary evaluation performed with a small sample of participants: the conclusion reached cannot be considered an absolute and immutable fact, meaning that more evaluation, adopting an empirical approach, is going to be mandatory in the future. More in detail, other than observing with more users the consequences of gamified mechanics, it will also be necessary to define an actual testing experiment, where evaluators would be tasked with testing pages containing actual bugs and issues, evaluating the issue-reporting functionality of the extension as well as the effectiveness given by game elements in locating said bugs. A more extensive evaluation on the support given to automated testing tools would also be required, with different test cases being adopted to understand whether the extension can generate a script that can faithfully reproduce a testing session, removing what is one of the challenges that make up GUI testing, the manual definition of script files. It is also important to note that the questionnaire may be subject to the so-called *Response Bias*: evaluators expect that the questioner is hoping to get positive answers and thus are led to answer more positively; the effect this bias can have may have influenced the results, which however remain positive, even if a bit influenced.

Chapter 5

Conclusions

It is important, after the results obtained during the preliminary evaluation and their confirmation of gamified mechanics as a valuable addition to the software testing process, to also assess the limitations found during both development and evaluation, in order to delineate a possible starting point for future development plans.

5.1 Limitations

The most concerning point of the extension, in terms of limitations found during development, consists in the generation of scripts compatible with SikuliX and Selenium. One concerning issue is the fact that it is not easy to distinguish page changes performed with browser actions (URL changes, forward/backward arrows), meaning that these actions cannot currently be identified and, as a consequence, also cannot be emulated by automated scripts. Regarding SikuliX some issues arise as a consequence of taking screenshots of widgets right after interaction:

- An element that appears only after hovering the mouse over another one (i.e. a menu containing two links that appears when placing the mouse cursor over the title of a web page) is screenshotted as-is, with no regards to the parent element; this causes a situation where the SikuliX script is unable to find the element to click since it does not appear unless the mouse has hovered correctly. A solution would be to screenshot the entire parent element and correctly place the offset, but it is unfeasible as there is no easy way to understand whether an HTML element has the property to show other elements when hovered on;
- When writing inside a text field the user might realize to have made an error, and would then rewrite the text content: such behavior is registered correctly

but is also hard to emulate, as SikuliX may need to find a match with the empty text field, with the text field with the wrong content and the text field with the content highlighted (the only way to rewrite content is to select all the content with the keyboard shortcut *CTRL + A* and then write the new content). Usually, the text field with content highlighted cannot be identified by SikuliX, leading to a failing test case;

- Clicking on a link does generally change its style, as links are intended to have a standard color and a color they assume when hovered on; the screenshot of a link is taken after a click, meaning that it is made with the hovering color rather than the default one and that a SikuliX script may not find a match, due to the different colors. To try and circumvent this issue, the extension takes an enlarged screenshot of the link, in hopes that giving SikuliX a larger area to find can reduce the chance of a script failing, but this solution is still prone to errors.

Another limitation of the extension comes from the highly customizable nature of web development: the fact that developers can choose to implement any kind of strategy when designing a page can lead to them selecting unusual practices (i.e. making a *div* element, usually used as a simple container, have a specific behavior when clicked, emulating what could easily be done with a *button* element). Such choices are obviously possible but cannot be predicted when defining the extension behavior, leading to the extension being able to support only elements that conventionally implement relevant actions (links, forms, buttons, dropdown menus); elements of a different type are not, at the moment, supported by the extension and thus are not included in the various game features (no progress bars, no count of elements found, no highlighting). The unpredictable nature of web development also means that elements with innate *Javascript* added (i.e. the use of an Event Listener defined by developers) are hard to predict and handle, as there is no way to know whether a given element actually has a listener attached to it or not, meaning that there is no way to find out if elements perform some specific code as a reaction to certain events. It is also possible that a page may have a subsection for which it is not possible to access the *DOM*, meaning that elements defined in the subsection cannot be detected by the extension, meaning that interaction with them is not recorded. Lastly, the evaluation sessions revealed that, if a website adopts a different layout for mobile viewing, the elements that make up said layout are counted as intended for coverage and score computations but, unfortunately, there is no easy way to understand which elements belong to the mobile layout or the desktop one. As a consequence, if a user does not know that the website under test has a mobile layout, the extension will show incomplete coverage values with no indication of how to access the missing elements.

Another issue, unrelated to features but more general, is that portability to

other browsers is not granted, due to different browsers adopting different policies in defining the APIs used for developing extensions. The browsers *Microsoft Edge* and *Opera* are naturally compatible with Chrome extensions, although the latter requires the installation of another plugin to ensure compatibility, but the same cannot be said about *Firefox* and *Safari*, which both use a different set of APIs. Making the extension compatible with *Firefox* would require a rewrite of the code, following the guidelines defined by **Mozilla**[20] for developing cross-browser extensions.

5.2 Future Plans

In light of the positive results obtained with the, albeit limited, evaluations, as well as the limitations just reported, future work on the extension is to be expected, in order to refine and solve the issues so as to create a useful gamified GUI testing tool. Future development plans should, ideally, focus on:

- Deploying a non-local server which would allow users from any location to use the extension, in case it would ever get published on the Google Chrome Web Store;
- Defining a more extensive method for identifying interactable elements by analyzing in detail the *HTML* document, to increase the support given to action recording. The definition of a strategy for identifying elements shown in mobile layouts, as well as a way to signal the existence of such elements, would also be needed;
- Increasing support for script generation, both in terms of generating script for more tools (eventually allowing users to choose for which tool(s) the scripts would be generated) and in terms of handling better actions performed to be replayed;
- Extending the gamified mechanics adopted: defining more achievements and avatars, with new methods to unlock them; extending the leaderboards, with the coverage one eventually being divided by different websites;
- Refactoring the code to be portable to other browsers without issues, increasing the eventual future user base;
- Performing a new evaluation session, which will involve a statistically significant amount of users, to formally evaluate the benefits that gamification can bring to GUI testing.

Bibliography

[1] RaiMan. *RaiMan's SikuliX*. URL: `http://sikulix.com/` (cit. on p. 5).

[2] SeleniumHQ. *Selenium IDE · Open source record and playback test automation for the web*. URL: `https://www.selenium.dev/selenium-ide/` (cit. on p. 7).

[3] Inc Eclipse Foundation. *Eclipse Scout - A one-stop framework to develop professional business applications*. URL: `https://www.eclipse.org/scout/` (cit. on p. 9).

[4] R. Feldt M. Nass E. Alégroth. «Augmented Testing: Industry Feedback To Shape a New Testing Technology». In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2019, pp. 176–183 (cit. on p. 9).

[5] K. Berkling and C. Thomas. «Gamification of a Software Engineering Course and a detailed analysis of the factors that lead to it's failure». In: *2013 International Conference on Interactive Collaborative Learning (ICL)*, pp. 525–530 (cit. on p. 13).

[6] T. Nash P. Anderson and R. McCauley. «Facilitating Programming Success in Data Science Courses through Gamified Scaffolding and Learn2Mine». In: pp. 99–104 (cit. on p. 14).

[7] A. Gambi G. Fraser M. Kreis and J. M. Rojas. «Gamifying a Software Testing Course with Code Defenders». In: *SIGCSE '19, February 27–March 2, 2019, Minneapolis, MN, USA*, pp. 571–577 (cit. on p. 14).

[8] V. H. S. Durelli H. M. dos Santos et al. «CleanGame: Gamifying the Identification of Code Smells». In: pp. 437–446 (cit. on p. 15).

[9] J. Brant M. Fowler K. Beck and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison- Wesley Professional, 1999 (cit. on pp. 15, 16).

[10] PMD. *PMD Source Code Analyzer*. URL: `https://pmd.github.io/` (cit. on p. 16).

[11] L. Hernandez M. Muñoz et al. «State of the Use of Gamification Elements in Software Development Teams». In: pp. 249–256 (cit. on p. 16).

[12] I. E. Ferreira Costa and S. R. Bezerra Oliveira. «A Systematic Strategy to Teaching of Exploratory Testing using Gamification». In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pp. 307–314 (cit. on p. 18).

[13] Google Chrome Developers. *What are extensions? - Chrome Developers*. URL: `https://developer.chrome.com/docs/extensions/mv3/overview/` (cit. on p. 20).

[14] gameart2d. *The Boy - Free Sprites - Game Art 2D*. URL: `https://www.gameart2d.com/the-boy---free-sprites.html` (cit. on p. 27).

[15] Stuk. *JSZip*. URL: `https://stuk.github.io/jszip/` (cit. on p. 36).

[16] Eli Grey. *Saving generated files on the client-side — Eli Grey*. URL: `https://eligrey.com/blog/saving-generated-files-on-the-client-side/` (cit. on p. 36).

[17] Software Engineering Research Group - Politecnico di Torino. *Software Engineering Research Group*. URL: `https://softeng.polito.it/` (cit. on p. 54).

[18] e-Lite Research Group. *e-Lite: Intelligent and Interactive Systems*. URL: `https://elite.polito.it/` (cit. on p. 54).

[19] B. A. Weerdmeester P. W. Jordan B. Thomas and I. L. McClelland. *Usability Evaluation in Industry*. 1996 (cit. on p. 58).

[20] Mozilla Foundation. *Building a cross-browser extension - Mozilla | MDN*. URL: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Build_a_cross_browser_extension` (cit. on p. 65).