Informatics

Enhancing Page Object Maintainability in Modern Automated Software Testing Life Cycles

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Christoph Hafner, BSc.
Matrikelnummer 01326088

an der Fakultät für Informatik
der Technischen Universität Wien

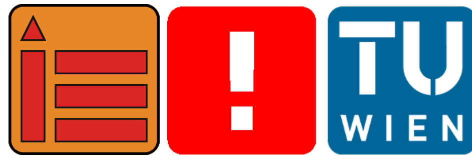
Betreuer: Thomas Grechenig

Wien, 08.07.2020

Unterschrift Verfasser

Unterschrift Betreuer

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



Enhancing Page Object Maintainability in Modern Automated Software Testing Life Cycles

Master's Thesis

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Christoph Hafner, BSc.
Registration Number 01326088

elaborated at the
Institute of Information Systems Engineering
Research Group for Industrial Software
to the Faculty of Informatics
at TU Wien

Advisor: Thomas Grechenig

Vienna, July 8, 2020

Technische Universität Wien, Forschungsgruppe INSO
A-1040 Wien ▪ Wiedner Hauptstr. 76/2/2 ▪ Tel. +43-1-587 21 97 ▪ www.inso.tuwien.ac.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Statement by Author

Christoph Hafner, BSc.
Wiedner Hauptstraße 73 / 2 / 12, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Place, Date) (Signature of Author)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

I would like to thank everyone who was involved in this thesis and made it possible. This especially includes Markus for reviewing the content and providing scientific answers on every question in almost no time and Matthias for proofreading my thesis and pointing out the same mistakes over and over again. Furthermore, great appreciation goes to all of my friends, especially my girlfriend Andrea, for motivating me throughout the whole process of writing this thesis and never allowing me to quit it. Lastly, I would like to thank my family, without their help neither this thesis nor my study would have been possible.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Das Testen von Software nimmt einen entscheidenden Teil des Software-Entwicklungsprozesses ein. Abhängig von der Art der geschriebenen Software werden verschiedene Testtechniken angewandt, um die Software bestmöglich zu testen. Das Testen von Benutzeroberflächen, als Teil von Systemtests, kann entweder automatisch oder manuell erfolgen. Während der manuelle Ansatz die Möglichkeit bietet, schnell auf Änderungen in der Benutzeroberfläche zu reagieren, erlaubt es der automatische Ansatz, die Benutzeroberfläche in verschiedenen Umfeldern ohne viel manuelle Arbeitsleistung zu testen. Um eine wartbare Testumgebung im automatischen Ansatz zu erhalten, wird das Page Object-Entwurfsmuster verwendet, welches eine Abstraktionsschicht zwischen dem Test und dem getesteten System formt. Da die Identifizierung von Web-Elementen jedoch sehr statisch ist und bereits durch kleine Änderungen in der Benutzeroberfläche falsch oder ungültig sein kann, müssen Page Objects während des Entwicklungsprozesses oft verifiziert und gegebenenfalls adaptiert werden, was in sich wiederholenden Aufgaben resultiert. Trotz verschiedener bestehender Programme, welche dieses Problem zu lösen versuchen, existiert keine Lösung, welche bestehende Page Objects wiederverwenden und die Verifizierung dieser automatisch durchführen kann. Basierend auf bestehenden Lösungen und sogenannten "Best Practices" der Software-Entwicklung werden in dieser Masterarbeit Anforderungen an ein Programm definiert, welches Tester in der Evaluierung und Adaptierung von Page Objects unterstützt und wiederkehrende Aufgaben reduziert. Um diese Anforderungen im Zuge einer Expertenevaluierung zu verifizieren und zusätzlich zu verbessern, wurde ein Prototyp entwickelt, welcher auf diesen Anforderungen aufbaut. Das Ergebnis dieser Evaluierung ist, dass, obwohl der Prototyp - aufgrund von nicht unterstützten Programmiersprachen oder Page Objekt Struktur - nicht für jeden Experten verwendbar ist, die Anforderungen bestätigt und akzeptiert wurden und großes Potential in einem Programm mit diesen gesehen wurde, besonders für Anfänger im Bereich des Testens von Software. Eine Adaptierung des Prototyps für weitere Programmiersprachen sowie eine bessere Integration in das Projekt werden in dieser Masterarbeit als zukünftige Schritte in diesem Themenbereich vorgeschlagen.

Schlüsselwörter

Software testen, Benutzeroberflächen Tests, Page Object Entwurfsmuster, Expertenevaluierung

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

// Abstract

Software testing has taken a crucial part in the software development life cycle in recent years. Depending on the type of software written, different approaches are used to achieve satisfying results. User interface tests, which are executed as a system test – i.e. testing the whole system at once –, can be achieved in either an automated or manual approach. While the manual approach allows for easy adaption to changes in the user interface, the automatic approach allows for easily testing the user interface in different environments regularly without the need for much manpower. In order to achieve a maintainable approach on automated tests, the page object design pattern is used, which provides an abstraction layer between the test and the system under test. However, as identification methods for web elements are very static and can be wrong or invalid even after small changes on the user interface, page objects need to be verified and potentially adapted often during the development life cycle, requiring a lot of repeating and time-consuming tasks doing so. While there are tools available which try to solve this problem, no tool allows for reusing existing page objects and verify their validity automatically. Based on existing state-of-the-art approaches and using best practices in software development, this thesis proposes requirements for a tool which supports the tester in the evaluation and adaption of page objects and tries to reduce the amount of repeating tasks. In order to verify the defined requirements and to refine them, a prototype taking the requirements into account is developed and used for an expert evaluation. The result of this evaluation is that, while the prototype is not ready to support every expert in their daily work – due to differences in programming languages or page object styles used –, they accept all the defined requirements and see a great potential in such a tool, especially for new or young software testers. An adaption of the prototype for further programming languages and a deeper integration into the project are future work tasks proposed in this thesis.

Keywords

software testing, user interface testing, page object pattern, expert evaluation

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

1 Introduction **1**
1.1 Problem description . . . 1
1.2 Aim of work . . . 2
1.3 Methodological approach . . . 3
1.4 Structure of the thesis . . . 4

2 Fundamentals **6**
2.1 Software testing . . . 6
2.1.1 Relevance of software testing . . . 6
2.1.2 Definition . . . 7
2.1.3 Test levels . . . 11
2.1.4 Test types . . . 12
2.1.5 Implementation approaches . . . 15
2.1.6 Frameworks . . . 16
2.2 Test automation . . . 19
2.3 User interface testing . . . 20
2.3.1 Fundamentals of user interface testing . . . 20
2.3.2 User interface element identification . . . 21
2.3.3 The page object pattern . . . 24

3 State of the Art **28**
3.1 APOGEN . . . 28
3.1.1 APOGEN's steps . . . 28
3.1.2 Tool evaluation . . . 30
3.2 Page Modeller . . . 30
3.3 Selenium Page Object Generator . . . 31
3.4 SWET - Selenium WebDriver Elementor Toolkit . . . 32
3.5 Further findings . . . 33
3.5.1 WTF PageObject Utility Chrome Extension . . . 33
3.5.2 PageObject-IO . . . 33
3.5.3 Selenium Code Generator . . . 33

4 Motivation and requirements **34**
4.1 Quality characteristics and their relevance for page objects . . . 34
4.2 Requirements . . . 35

5 Prototype concept **38**
5.1 Prototype characteristics . . . 38
5.1.1 Application type . . . 38
5.2 Expected usage behavior and mock-ups . . . 40
5.2.1 Toolbar . . . 40
5.2.2 Project area . . . 41
5.2.3 Page object area . . . 42

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6 Implementation 44
6.1 Implementation basics 44
6.1.1 Chrome extension basics 44
6.1.2 WebExtension API Polyfill 47
6.1.3 Selenium 48
6.2 POGito 49
6.2.1 POGito in general 49
6.2.2 Code structure 51
6.2.3 Selected implementations 53

7 Evaluation 60
7.1 Evaluation process 60
7.1.1 Goal 60
7.1.2 Participant characterization 60
7.1.3 Evaluation environment 60
7.1.4 Methodology 61
7.1.5 Scenarios 65
7.2 Threats to validity 70
7.3 Evaluation results 71
7.3.1 Results of the pre-demonstration questionnaire 71
7.3.2 Requirement relevance 72
7.3.3 Feedback on the prototype and its functionality 73

8 Conclusion 76
8.1 Recap 76
8.2 Future work 77
8.2.1 Better source code integration 77
8.2.2 Wider range of programming languages 77
8.2.3 Method maintenance 77
8.2.4 Support for different page object formats 77
8.2.5 Mobile support 78

Bibliography 79
References 79
Online References 81

A Appendix 84
A.1 Interview 84
A.2 Scenario-Outline 87

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

2.1 Error-Fault-Failure adapted from [60] . . . 8
2.2 Test-process as shown in [72] . . . 9
2.3 Four levels of software tests [32] . . . 13
2.4 Visualization of a usual testing approach (left) and a model-based testing approach (right) . . . 15
2.5 Sequencediagram of running a JUnit test with Selenium . . . 19
2.6 The test automation pyramid as described in [16] . . . 20
2.7 Break-Even point of manual and automatic software testing as shown in [57] . . . 21
2.8 The DOM tree displayed as graph . . . 22
2.9 Page objects in relation to test scripts . . . 24
2.10 Web page (left) and its corresponding source code (right) . . . 25

3.1 High-level overview of APOGEN's approach for web page object creation as shown in [73] . . . 29
3.2 Screenshot of Selenium Page Object Generator . . . 32

4.1 Overview of the requirements' origin . . . 37

5.1 Basic user interface parts . . . 39
5.2 Mock-up of the toolbar on top of the application . . . 40
5.3 Mock-up of the project-related view on the left of the application with an opened drop-down menu for further actions . . . 41
5.4 Mock-up of the page object view allowing modification and verification with an opened drop-down menu for further actions . . . 42
5.5 The full user interface as planned . . . 43

6.1 Screenshot of a web page in normal state (left) and during the highlighting of an element in the top right corner (right) . . . 49
6.2 Screenshot of POGito after verification of the paths, resulting in three valid and one invalid variables . . . 52
6.3 Page object in POGito before the export . . . 56
6.4 Screenshot of POGito (left) and the browser (right) during adding a new variable . . 57

7.1 Screenshot of the bugstore . . . 61
7.2 Scenarios and the defined requirements they address . . . 70
7.3 Relevance for the given feature between 1 (not relevant) and 4 (very relevant) . . . 73
7.4 The mean value of the answers given by the experts on the defined question from 1 (most negative option) to 4 (most positive option) when asked about POGito . . . 74

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

List of Tables

2.1 Project size and typical error density [48] 6
2.2 Side-by-side comparison of different syntax [87] 23

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Listings

2.1 Example JUnit testclass (simplified) 17
2.2 The DOM tree serialized as text 22
2.3 Example page object 26
2.4 Example JUnit 5 test using page objects 27

6.1 Excerpt of POGito's manifest file 45
6.2 Sample background script method [24] 46
6.3 Sample messaging between extension and content script [49] 46
6.4 Usage of content scripts as shown in [17] 47
6.5 Sample messaging between content script and extension [84] 48
6.6 Sample code of the highlighting of an element as used in POGito 50
6.7 Package structure of POGito (not complete) 53
6.8 Sample code of the export function as used in POGito 54
6.9 Handlebars template used for Java export 55
6.10 Exported page object of POGito 56
6.11 Sample code of the select process for a new varibale 58
6.12 Excerp of the TargetSelector (in targetSelector.js) 59

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

1 Introduction

This chapter introduces the master thesis by giving an overview of problems that could be encountered when using the page object pattern and how these problems could be approached. Furthermore, the methodological approach is presented, based on which this thesis tries to solve the problems. Finally, an overview of the chapters in this thesis is given.

1.1 Problem description

In modern-day software development, testing the software is a crucial part in the development life cycle. Due to the size and complexity of applications, testing is mostly done in an automated fashion using frameworks and tools supporting the activity. Depending on the type of software written, different approaches are used to achieve satisfying results. For example, backend components can be tested on a low level (i.e. methods). User interfaces are often tested by simulating the user's behavior either by capture and replay or by defining steps programmatically.

Independent of the chosen development process, software is changed frequently, especially during the development phase. When following the Scrum process, a de-facto standard in agile software development for applications [26], the application gets changed in most cases at least bi-weekly, resulting in the need of adaptions of existing tests and writing new ones for newly implemented features. While software tests on a low level are implemented calling the methods directly, high level tests focusing on interactions with the user interface often have to deal with a very volatile component. Depending on the chosen approach for accessing elements on the screen, even the smallest change can result in a failure of one or more tests. If the test is using the coordinates of an element, these coordinates can easily be off when the element gets shifted, for example, if a predecessor label is added. Another way for accessing the elements is by using the DOM tree which is further explained in chapter 2. When working with this approach the element is accessed via its path on the DOM tree, which again can easily be shifted, for example, if the element gets wrapped into another element. As before, the test then fails even though the element is present. Properly creating and maintaining user interface tests, therefore, requires a lot of resources. However, as these tests can also be seen as system tests (explained more deeply in chapter 2) the relevance of these tests is also high and therefore not neglectable.

As the number of software tests in a test suite can get very high on large-scale applications, different testing principals can be used to reduce the work needed for adaptions of the tests. One principal for such volatile environments are model-based tests, where an abstraction layer is built between the test and the system under test. This abstraction layer then can be adapted to the new environment without the need of adaptions on the tests. When testing user interfaces, the page object pattern is often used for implementing model-based tests on user interfaces in various applications like native ones, as well as together with Selenium and their WebDriver-API [69] for simulating the user's behavior in a web browser. By doing so, page objects are expected to provide a maintainable approach on writing tests for web applications. Another benefit of them is that they are mostly written in the same programming language as the tests [44] and provide access to the page's elements in the DOM tree. Identifying the page's elements uniquely, however, is often difficult to achieve or even impossible and dynamic approaches reach their limits doing so. Therefore, static approaches are often chosen instead. Static approaches allow satisfactory results on static pages, but can also result in false positives or failure due to the volatility of the DOM tree in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

more dynamic environments. Even small changes on the user interface can result in a completely unusable page object.

Along with adaptions of tests validating the application's code, page objects have to be maintained after each development cycle as well. Besides writing new page objects for newly created pages, existing ones need to be validated and manually updated in case of changes. Even though several tools exist which try to tackle the problem, none of them are capable of actually maintaining the page object. When using these tools, page objects either require a lot of manual action by the user in each iteration or produce a lot of boilerplate or dead code. As this code is in most cases also maintained and tracked in version control systems like Git, the history also gets easily polluted by this unnecessary code, resulting in harder traceability of changes.

1.2 Aim of work

The aim of this work is to improve the creation and maintenance process of page objects in the software testing life cycle by providing a tool-supported, semi-automated approach. The main benefit of this approach, in contrast to existing ones, is the reusability of existing page objects throughout the life cycle by providing import and export capabilities. Furthermore, the user has visual support to spot invalid or moved variables of a page object easily along with support to add and modify variables on the page object. In addition, a prototype is implemented, which is used for the evaluation of the suggested approach.

The mentioned evaluation of the approach is conducted by experts in this field, using expert interviews. Besides an assessment of the defined requirements, the interviews shall also provide insights regarding further problems software testers face when maintaining page objects.

In order to achieve the aim of this work, the following research questions shall be answered:

- **RQ1: What are existing solutions for page object generation and how do they support the developer?**

  This question shall be answered by conducting a literature and state of the art research of available solutions in this research field. By doing so, the benefits and drawbacks of available solutions shall be found in order to define requirements for a system enhancing them. This research question will be answered in chapter 3, which focuses on the state of the art of page object generation.

- **RQ2: What are the main requirements and a suitable design for a system that creates and maintains page objects in a semi-automated way?**

  The focus of this research question is to define requirements on a system improving the creation and maintenance process of page objects. Furthermore, a prototype is designed taking the requirements into account. For the requirements as well as for the design, best practices of already available solutions shall be taken into account and enhanced by general best practices for supporting the maintainability. This research question will be answered in chapter 4, focusing on the requirements analysis and chapter 5, focusing on the concept of the prototype.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

- **RQ3: Can the defined requirements enhance the current situation for software testers and what are the benefits and drawbacks of the proposed approach in productive environments?**

  Based on the developed prototype, the defined requirements are appraised using an expert evaluation. This research question shall be answered based on the results of the interviews with the experts. Furthermore, shortcomings and directions for future work on the requirements and prototype shall be received from the answers. The discussion and outcome of this research question can be found in chapter 7 which discusses the results of the expert evaluation.

Along with the research questions which shall be answered in this thesis, the following hypothesis is set and shall be answered after the expert evaluation:

> The developed prototype, which is based on the defined requirements for a tool-supported semi-automated approach for page object maintenance, improves the testing process of volatile web applications by reducing the work needed for maintaining the model objects of the tests. By doing so, automated regression tests can be performed more efficiently and less error-prone as opposed to manual maintenance.

1.3 Methodological approach

This section describes the chosen methodological approach for writing this thesis. The following steps are executed in the order mentioned to follow a scientific methodology:

Literature and state of the art analysis

In the first step, a literature research was conducted in order to find existing solutions for the problem. As one publication was already known, the snowballing method using forward and backward snowballing as described in [42] was initially used for finding additional scientific literature. As this showed that research on page object maintenance is not very broad, the research technique switched to a general keyword-based search. Based on the results of this research, additional tools could also be found. The combined results of all research show that a variety of tools exists, but many of them are deprecated or have not been maintained for years. Furthermore, none of the found tools follow a maintainability approach for their page object generation.

Requirement analysis

With the information gathered during the research and from available solutions, requirements for introducing a maintainability approach to the field of automatic page object generation were defined. These requirements were based either on functionality provided by existing tools or based on best practices for areas no tool provides functionality in. In addition to functional requirements, non-functional requirements were also defined.

Design & Conception

Using the defined requirements, a concept for a prototype was developed. Based on that, experts in the field shall evaluate the requirements for their meaningfulness.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Implementation

Based on the concept defined in the previous step, a prototype was implemented. This prototype was developed based on a commonly known Capture & Replay tool called Selenium IDE. Furthermore, the prototype was called "POGito" – using the first letters of Page Object Generator followed by the ending -ito stemming from the popular testing tool mockito – to better refer to it in this thesis and during the interviews. The developed prototype tries to solve the previously defined problems which experts have to deal with.

Evaluation

In order to answer the research questions and hypothesis defined previously and to evaluate if the defined requirements really address the experts' daily problems, an expert evaluation was conducted. For this evaluation, an expert interview was chosen as an approach to gain the experts' opinions using both quantitative and qualitative questions in the questionnaire. This mix allows for gathering data about the thoughts of the experts but also allows for gathering statistical data across all experts. Before the interview was conducted, a demonstration of the main use cases using the developed prototype was done, giving them an overview of capabilities a tool with the defined requirements can have. The given demonstration was intentionally not done by the experts themselves but rather by the interviewer, as the questionnaire shall only be answered based on the experts' thoughts of the functionality rather than the usability of the prototype.

1.4 Structure of the thesis

This thesis describes the development and evaluation process of a prototype application, solving the problems described above. In order to understand the underlying chapters, chapter 2 gives an introduction to software testing in general, automatic software testing as well as page objects and how they are used in the testing process.

Based on the literature research conducted, the following chapter 3 gives an overview of the results found. In this overview, besides a description, the expected advantages and drawbacks of the solutions are discussed.

Following the literature review, a requirement analysis is done in chapter 4. The expected outcome of this are requirements for solving the defined problem of maintainability in the page object generation. These requirements are on one hand based on already available solutions with their advantages and drawbacks and on the other hand specific requirements for solving the described problem.

Based on the defined requirements, chapter 5 will then present the concept for a prototype implementing them. Besides a general discussion on the implementation approach, mock-ups will be presented and the expected usage behavior described, based on which a prototype shall be implemented.

Chapter 6 then presents the developed prototype, discusses the technology stack and includes source code snippets for better comprehensibility. Furthermore, screenshots of the tool will be provided to visualize the experience. Additionally, background information on the development process and problems occurred are described and limitations of the prototype known at this stage presented.

In order to verify the defined requirements, an expert evaluation is conducted. The process and the results of this evaluation are presented in chapter 7, along with questionnaires used for the expert interviews. Besides the direct results of the questionnaires, the newly acquired knowledge, as well as new limitations found through the evaluation, are also presented and discussed.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 8 concludes the thesis by giving a recapitulation of the process and results. Additionally, future work on both the research part and the prototype are presented.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2 Fundamentals

In this chapter, the fundamentals of this master thesis are explained. Therefore, section 2.1 focuses on software testing in general, its definition and relevance in the development life cycle. Furthermore, different testing levels, test types and implementation approaches are described in this section along with two frameworks relevant for this thesis. Section 2.2 then focuses on the automation of software testing and describes why testing software should be automated. The chapter ends in section 2.3 by describing and discussing the fundamentals of user interface testing along with how elements in the user interface can be identified in automated tests. Additionally, the page object pattern is described in this section, giving an introduction on what the pattern is used for and how it can be used for testing web applications.

2.1 Software testing

This section focuses on the fundamentals of software testing. Software testing has long been seen as a duty demanding high resources but has evolved to be a fixed part of the development life cycle and is also seen as an expert task [29] [91]. Therefore, establishing good testing skills should begin as early as possible when developing programming skills [15].

2.1.1 Relevance of software testing

Software has taken a crucial part in modern life and can be found almost everywhere today. While software can enhance the experience of a product, it can also lead to immense problems if behaving unexpectedly. Errors in software may result in a completely unusable state of the application, sensitive data being lost or stolen, or potentially even death of people. One of the most discussed software problems in recent history is the story of Boeing. Even though it also included human fault, the software was responsible for the death of 346 people and a standstill of all Boeing 737-Max for several months [52].

With even more upcoming products, which depend heavily on software, like self-driving cars, software is aimed to contain as few errors as possible not to harm anyone by taking wrong decisions based on wrongly developed software. On the other hand, as pointed out by McConnell [48], when software is getting bigger the number of potential software errors also rises as shown in table 2.1.

Project size (in lines of code)	Typical Error Density
Smaller than 2K	0-25 errors per thousand lines of code (KLOC)
2K-16K	0-40 errors per KLOC
16K-64K	0.5-50 errors per KLOC
64K-512K	2-70 errors per KLOC
512K or more	4-100 errors per KLOC

Table 2.1: Project size and typical error density [48]

According to statistics by Doughty-White, an average modern car in 2015 had about 100 million lines of code [20]. Considering that a program with 512 thousand lines of code already contains 2.048-51.200 errors, an average modern car would contain more than 600.000 software errors. Of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

course, this can only be seen as a rough calculation based on the mentioned numbers and without any quality assurance. Furthermore, not every error is leading to a problem, however it shows that testing software is crucial to reduce the number of errors and the chances to harm either the car itself or any person on the inside or outside.

Another sector of software testing becoming more and more important is security testing. According to Potter and McGraw "standard software testing literature is concerned only with what happens when software fails, regardless of intention" [56]. In contrast to testing the correct execution of code, testing software security "is about making software behave correctly in the presence of a malicious attack" [56]. This also includes preventing leakage or modification of data saved on the system by the user or other information leakages like source code [82]. Even model-based testing (described in 2.1.4), the testing technique this thesis focuses on, can be conducted on the security side, as discussed in [22].

Software testing does not only include testing parts of a program that are executed in the background (e.g. the software's code) but also access points to these parts for an external user. This user could either be a program again or a human on a terminal or a graphical user interface. User interface tests ensure that interactions by a third party are handled as expected and the user is able to interact with the application as they should. This includes testing for the presence of elements on the user interface, that they are displayed correctly and the application is controllable with respect to usability.

In general, software testing is intended to gain confidence about the quality of the developed application in every aspect of its existence, like behavior, usability and security. As already shown in table 2.1, it is impossible to write software without any error, but not every error will lead to a problem for the user and most of the errors remain hidden for a long time as their impact is very small [89]. However, the goal of software testing is to find as many of these problems as possible to reduce the risk of a problem occurring in a productive environment either noticed or unnoticed by the developer, but with respect to resources like money, manpower or time.

2.1.2 Definition

Besides the relevance software testing has gained in the software development process, it has also been defined in literature. When conducting a literature research, several different definitions can be found, but all have in common that software testing belongs to the field of quality assurance [60] [55] [32]. The Institute of Electrical and Electronics Engineer (IEEE) defines in their standard glossary of software engineering the process of software testing as follows [34]:

> Software testing is a formal process carried out by a specialized testing team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer. All the associated tests are performed according to approved test procedures on approved test cases.

In this definition, IEEE requires a formal process, approved test procedures and test cases for software testing. In order to comply to this definition, testing has to be planned and executed over the whole process of developing the application, but in a formal way. Grechenig, however, suggests a combination of formal and informal methods for finding bugs [29]. While formal methods use a defined setting and execution, informal methods are executed based on intuition of the tester [29].

The International Software Testing Qualifications Board (ISTQB), an international software testing certification board, provides a standardized qualification for software testers which also defines software testing but in a different way to the previous one [9]:

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

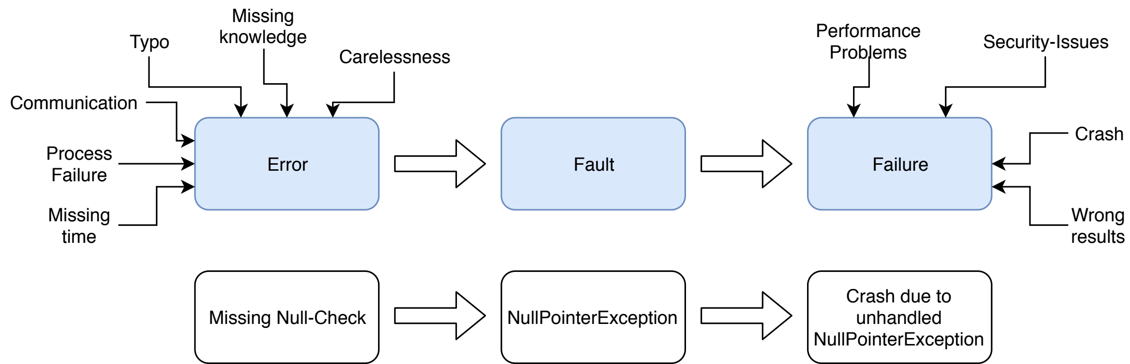


Figure 2.1: Error-Fault-Failure adapted from [60]

> Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.

In contrast to the definition of the IEEE, the ISTQB includes software failures in their definition. A software failure is the end result of a mistake made by a developer. Figure 2.1 shows how a mistake (error) can lead to a failure and extends the figure from [60] by an example. It should also be noted that according to Yadav, Yadav, and Verma not every defect will necessarily result in a failure as erroneous code might not be accessed by the user [89].

While the previous two definitions were a more formal way to describe software testing, Myers and Sandler define software testing short and precisely [51]:

> The objective of testing is to prove that there are bugs [..]

As already discussed in table 2.1, every software contains bugs. The definition from Myers and Sandler confirms this. This standpoint and the fact that testing everything is impossible – resulting in finding a chance to find a bug everywhere – are two of the seven testing principals of the ISTQB, which should provide the tester with so-called best practices in software testing [28] [72]:

1. **Testing shows the presence of defects, not their absence**
   When testing software, defects can be found, and the number of defects can be reduced. But even in case no test detects a defect anymore, it cannot be seen as proof of total correctness.

2. **Exhaustive testing is impossible**
   Testing the whole application, meaning all combinations and inputs, is mostly not achievable. Setting and focusing on priorities and using "best practice" is sufficient to achieve a good confidence about the program.

3. **Early testing saves time and money**
   Testing the software early and regularly results in finding and fixing errors as soon as they arise. Fixing errors in later stages or even in production often results in high costs. Techniques like Test-Drive-Development (TDD) have become popular in recent years, where tests are written before the corresponding code is written [37].

4. **Defects cluster together**
   "A small number of modules usually contain most of the defects discovered during pre-release testing or are responsible for the most operational failures" [28]. Therefore, it is often good to do a risk analysis in order to detect clusters of defects more easily [72].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

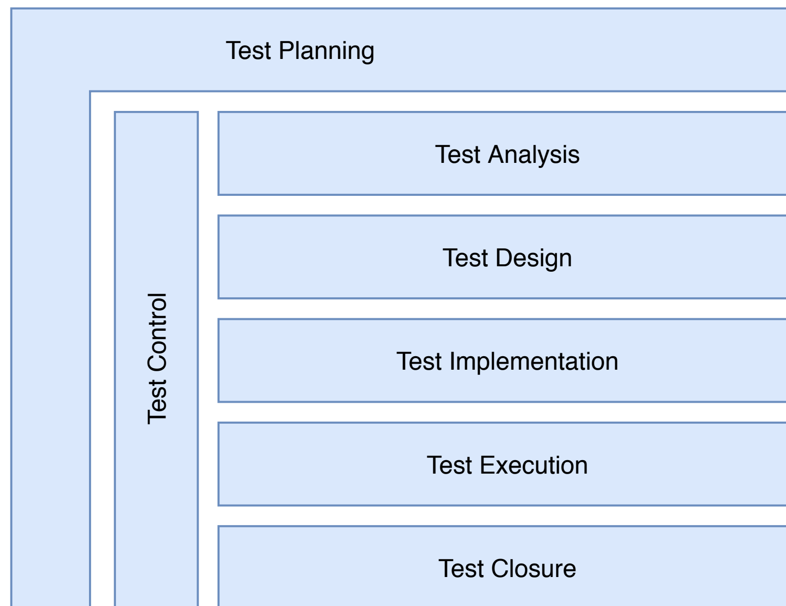


Figure 2.2: Test-process as shown in [72]

5. **Beware of the pesticide paradox**
   Testing the same code by using the same tests multiple times does not result in finding any more defects. Test cases should be maintained, adapted and recreated in order to find new defects regularly.

6. **Testing is context depended**
   Different software needs to be tested differently. Therefore, the testing process needs to be adapted to fit the specific project.

7. **Absence-of-errors is a fallacy**
   Even though the system is well tested, this does not mean that there are no errors, or the system is perfect. Well tested software can still be unusable due to missing usability or not meeting the customer's needs [72].

The last one of the testing principles – number 7 – focuses on a common mistake. Software testing is often seen as only writing and executing test scripts. However, tests should not only cover if the program is running without failures, but also if the product is usable and built right according to the requirements of the customer. Testing the program against the requirements is called verification, while testing if the product is built as expected by the customer is called validation [55]. Both tasks have to be planned in a planning phase of the software test and analyzed afterwards. This shows that testing is rather a process than single activity which is wrapped around the whole software development life cycle. Without common sets of test activities forming a test process, software testing might not achieve the objectives according to ISTQB [9]. Spillner and Linz also mention that the testing process needs to be adapted to the project's needs, considering test levels and types as well as available resources [72]. The ISTQB fundamental test process, depicted in figure 2.2, shows the different activities of the process and how they belong together. It should be noted that the activities can also take place concurrently or even iteratively in certain development processes [59].

The following list briefly describes the objective of each part shown in figure 2.2 according to [72], [59] and [9]:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Test Planning**
  The test planning phase is the first one, started together with the project. It defines a test concept containing a definition of the test strategy, test objects and test procedures as well as an effort estimation and role allocations. Additionally, a test plan is developed, containing all test tasks with prioritization and due dates. During the development phase, the mentioned artifacts are monitored and, if needed, adapted.

- **Test Control**
  Like the previous stage, this stage happens over the whole development life cycle. The test control is responsible for starting, monitoring and reporting tasks defined in the test plan and taking steps in case of problems occurring. These activities also include the measuring of the quality of the tasks based on developed metrics. If some defined end criteria cannot be met, additional tests have to be developed and executed. Furthermore, the progress using the defined metrics is reported to stakeholders regularly.

- **Test Analysis**
  This phase is used to analyze all the information available for the future project. Therefore, documents like specifications and use case descriptions, design and implementation information like diagrams, or already available code is analyzed to determine test objectives. Furthermore, features of the application are identified to define tests for them. When creating the tests, a bidirectional traceability shall be available such that one can trace the test back to its requirements and vice-versa.

- **Test Design**
  The test design phase uses the knowledge of the previous stage to define concrete high-level test cases and other test ware. While the previous stage was about "what to test?", this stage defines "how to test?". Artifacts of this stage are designed and prioritized test cases as well as required data for them and a design of the test environment with required infrastructure and tools.

- **Test Implementation**
  This phase often takes place simultaneously to the test design stage or is combined with it into one. Most of the previous tasks are finalized and verified in this stage, for example the test ware - the underlying support structure of the tests. In this stage, test procedures are developed and prioritized and automated test scripts are written. ISTQB [9] describes the question to be answered in this stage as "do we now have everything in place to run the tests?"

- **Test Execution**
  In this stage, the defined tests are executed either manually or automatically, depending on the implementations of the tests defined in the previous steps. Each execution and its results are documented including a unique ID, versions, tools and test ware to trace each test. In case of a failure, found defects are reported back to developers and, after changes, old tests repeated or adapted. This might also include confirmation testing or regression testing as part of the whole test life cycle.

- **Test Closure**
  This stage concludes the testing activities and archives the data gathered in the testing life cycle. This happens at regular intervals, e.g. at a release, when a project milestone is reached or in an agile project after an iteration. This stage also includes a retrospective in order to improve the test process by analyzing the previous test activities.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2.1.3 Test levels

Following the above-defined process for testing, the implementation and execution activities can be further broken down into different test levels. These levels do not only define which part of the software is being tested but also their temporal relation to each other [32]. Furthermore, the effort required for the implementation, maintenance and execution increases in each test level. According to Hoffmann each test case can be defined in one of the following four test levels [32]:

- **Component or Unit Test**
  Component tests are used to test the smallest elements of a program. As this kind of test evaluates the methods and other isolateable sections of a program independently of each other, component tests are often referred to as Unit tests - testing a single unit of software. External components like other classes are mocked for these tests to provide expected responses. Component tests then evaluate if the method is behaving as expected by calling the method with predefined parameters (if there are any) and comparing the outcome of it with the expected outcome. There is a wide range of results which can be evaluated depending on the functionality a method provides, like the return value of the method or the number of invocations of other methods.

  This kind of software test is done by developers at the stage of writing the component. Depending on their developing approach, these tests are either written before (Test-Driven-Development [37]) or after the component is finished. Due to their size, these tests – or at least a subset of them – can be executed regularly on their own device.

- **Integration Test**
  When several components are finished and each has been tested on its own, those components are connected. An Integration Test then makes sure that connected components still work together as expected. Hoffmann [32] describes the following three strategies for integration:

  – **Big-Bang-Integration**
    In this integration strategy, all components are integrated at once. While the advantage of this strategy is that all connections are available at once and no component has to be simulated (mocked), it also has the downside of harder localization of problems occurring in the connections. Another downside is that all components need to be finished in order to integrate them.

  – **Structural Integration**
    The structural integration focuses on the structural dependencies between multiple components. If there are, for example, 4 components A->B->C->D depending on each other in this way, they are integrated in a defined order. This order is chosen out of 4 possible structural strategies:

    ∗ **Bottom-Up-Integration**
      This integration starts with the lowest components which do not have a connection to any other component. In the defined case component D would be the first component to be implemented and integrated, followed by C, B and finally A.

    ∗ **Top-Down-Integration**
      This strategy is the opposite of the Bottom-Up-Integration. Component A would be the first to be integrated, followed by B, C and finally D. The advantage of this approach is that for the end user the base is already available while different components are added throughout the life cycle. A downside of this approach is that missing components have to be simulated until they are integrated.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

  * **Outside-In-Integration**
    The Outside-In-Integration approach combines Bottom-Up- and Top-Down-Integration. In this case, A and D would be integrated first, followed by B and C. This again has the advantage that the end-user has a first prototype available while the developer does not have to simulate the lowest components - usually containing the most logic, as they are providing the data.
  * **Inside-Out-Integration**
    This approach is again the opposite of the Outside-In-Integration. It is the least used approach of the mentioned ones, because it does not provide any advantage not provided by the others but brings the downsides of Bottom-Up- and Top-Down-Integration with it.
- **Function-Oriented Integration**
  Using this integration strategy, functional criteria are used for integrating several components with each other. Mentioned strategies in this category are schedule-driven (components which come first, are integrated first), risk-driven (those components with the highest risk of interfering in the connection are integrated first), test-driven (based on a given test-case, required components are integrated first) and use case driven (components belonging to a common use case are integrated first).

- **System Test**
  On the system test level, the whole system is used and tested as one, throughout all of its layers. In this test level, it is checked that the system works as expected and corresponds to the specification. The test environment should resemble the productive environment in order to verify that the software works as expected. However, finding the cause of the problem or debug the program at this stage gets increasingly difficult, which requires the components to be tested well before.

  Furthermore, it should be noted that the test perspective of this level is from the users' standpoint, interacting through a provided interface with the application. Therefore, user interface tests – on which the focus of this thesis lies – are executed on this test level [72]

- **Acceptance Test**
  This test level is the final stage before the system is brought into production. In this stage the software is already tested by the customer and is executed on the customer's environment with real data provided. This level is also relevant for legal reasons regarding expectations and specifications of the customer. In contrast to system tests, the goal here is not to find problems of the implementation, but, according to ISTQB, to verify if the system is accepted for production.

Figure 2.3 shows the four test levels in their temporal and structural relation.

2.1.4 Test types

Testing software can be achieved using different approaches, referred to as test types. Many test plans often consist of a mix of different types in order to find problems one approach alone might not have found. Test types in general can be divided into two main categories: Software can be tested either statically (using code reviews, code walkthroughs, code analysis tools, etc.) or dynamically, by executing an instance or parts of a program and running different tests on it [45]. As this thesis focuses on user interface testing, a dynamic way of testing the application, only dynamic test approaches relevant for user interface testing will be described. In the following, the four test types that are most relevant for this thesis are discussed, while the main focus lies on the last one - model-based testing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

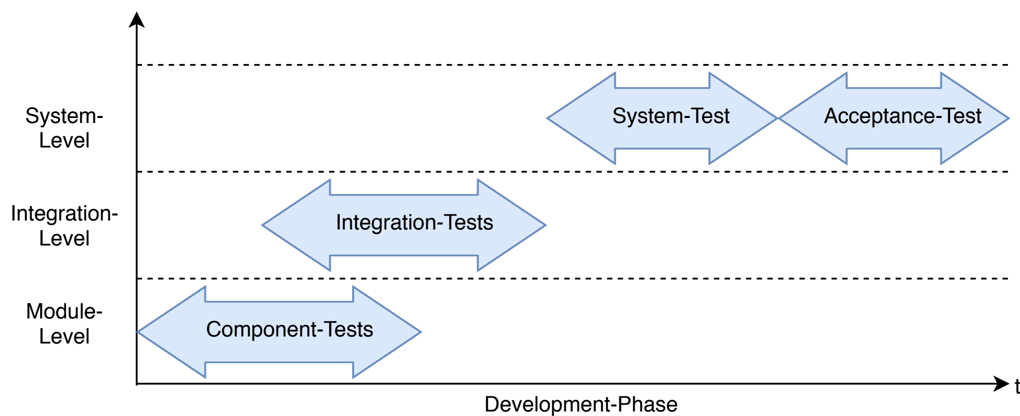


Figure 2.3: Four levels of software tests [32]

Manual testing

As the name suggests, this is the least automated approach and executed by a tester manually. When executing manual tests, the tester might use a predefined plan and step definitions for executing the tests, to have reproducible results, or just interact with the application exploratively. While this kind of test can be executed mostly without a setup needed, the downside is that it is a time-consuming approach, as one or more tester are needed for execution. Another downside of this approach is that repetitive executions of a test run require the same amount of resources each time and can get very boring over time. [77]

Script-based testing

When using the script-based testing approach, test scripts are developed which execute the application and verify its behavior. Most script-based tests are executed on lower levels of the application, e.g. the methods, but there are also script based tests for user interfaces like Espresso [1] for testing Android user interfaces. The main benefit of this approach is that tests can run automatically and fast either on local devices or in a build pipeline as already described before. Drawbacks of this approach are the prior knowledge required by a tester to write tests, and the resources which are required for creating and maintaining test scripts. Using this approach, the test scripts also need to be adapted once requirements or implementation details change. As the number of test scripts rises with the size of the application, their maintenance can be very time consuming and require a lot of resources. Therefore, good abstraction of the tests is the key to reduce maintenance costs. [77]

Capture-Replay testing

The capture-replay approach is a combination of the before mentioned manual testing and script-based testing approaches. In the first phase, the tester interacts with the application as they would do in a manual test. However, their behavior is recorded by a tool and later translated and exported to test scripts [72]. These test scripts can be executed automatically as mentioned before in script-based testing, or simply be interpreted by the same program used for recording it for execution as well. While this approach has the benefit of not requiring the user to have any knowledge in a programming language, there are several downsides of this approach: This approach again requires a lot of resources, as tests need to be re-recorded or adapted as soon as small changes in the user interface are made. Another downside can be the quality of the test scripts as they heavily depend on the export of the tool. This might result in code duplication or higher resource requirements due to bad test cases. [77]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Model-based testing

Using the model-based testing approach, models are used to assist the testing process by generating test artifacts automatically. Marciniak [47] is cited in [59] describing model-based testing as follows:

> Model-Based Testing (MBT) is [...] an approach that bases common testing tasks such as test case generation and test result evaluation on a model of the application under test.

By shifting the logic and data to a model, the test cases which are executed on the application can easily be changed by changing parameters in the model. This allows the tester to easily adapt the test cases even in volatile applications, saving a lot of resources and a significant speedup in testing.

Roßner describes in [59] three different approaches on how an application can be tested using model-based testing:

- **System-Model-driven**
  Using the System-model-driven approach, the system and its behavior are modeled using a modeling language like UML. Based on that system model, tests can be generated and exported. As a main benefit here, the single model can be seen, which can be used for code and test generation. However, this approach comes with the downside that if the model contains an error, the tests will not find any error in the code as they are based on the same definition.
- **Test-Model-driven**
  When using this approach, modeling languages like UML 2.0 Testing Profile (U2TP) or Test Control Notation (TTCN-3) are used for modeling test cases or test environments. Based on the modeled environments tests are generated automatically. According to [7] UML models can even be seen as software program from a tester's perspective. One advantage of U2TP, for example, is the reusability of already generated UML system models. Both modeling languages mentioned provide a graphical representation of the modeled test. Another way to transform models to actual test cases are from formal models like Finite State Machines, which generation process to JUnit tests is discussed in [30].
- **System- and Test-Model-driven**
  This approach is a combination of both previously mentioned ones. Depending on the preference, the focus can be laid either on the system side, the testing side or equally on both. As a benefit of this approach, the two independent models can be seen. If one contains an error, the chances are high that the other one detects it. However, this can also be seen as a drawback, as two models have to be maintained, meaning higher maintenance costs [59].

Furthermore Roßner defines three model categories used in model-based testing:

- **Environment models**
  It represents the environment the system shall exist and work in. It models characteristics of the environment like its behavior or limitations and can also include its physical conditions [59].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **System models**
  They describe the modeled system, the components of it and how they interact with each other. According to [59], system models are mostly used on a lower level like component or integration tests and can also be used to generate tests. They further provide an overview of the inner structures of the system [59].

- **Test models**
  This category is mentioned to be the most important one in model-based testing in [59] where the authors further explain that a test model "describes the system characteristics as well as generates or creates tests or their properties" [59]. Test models can further be broken down into test-base model and test-specification model. While the focus of the test-specification model lies on the tests, the test-base model's focus lies on relevant parts of the test base like structure and behavior. Zoffi mentions that the test-base model provides an abstraction layer between the tests and the system under test (SUT) allowing to easily adapt the model after changes in the SUT as also shown in figure 2.4 [91].

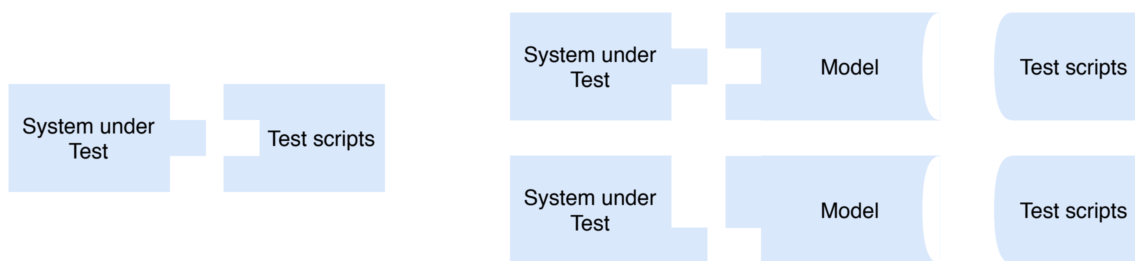


Figure 2.4: Visualization of a usual testing approach (left) and a model-based testing approach (right)

2.1.5 Implementation approaches

When implementing and executing tests, the application which shall be tested - often referred to as system under test (SUT) - can be seen in different ways. This subsection shortly describes three different viewpoints on an application during the development and execution period of tests.

- **Black-Box tests**
  Black-Box tests are created and executed without knowing the characteristics of the software's internal behavior. Tests in this category have to be generated based on the specification, as no other characteristics are known to the tester [32]. In order to reduce the number of required tests, Hoffmann mentions that using for example equivalence partitioning or boundary value analysis allows for narrowing down the possible inputs. It should also be noted that user interface tests are mostly executed as black-box tests, as the user interfaces hides the internal functions of a system.

- **White-Box tests**
  Using White-Box testing, the internal behavior is known to the tester. Tests in this category are mostly done by the developer itself, knowing the code and structure or by a tester who made himself familiar with it. This allows for testing with the knowledge of the characteristics in mind. The authors of the mentioned publication differentiate between control-flow testing or data-flow testing in this category. [32]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Gray-Box tests**
  This test category is a combination of Black-Box and White-Box tests. Without a formal process specified, the tester gains an overview of the software code and writes its Black-Box tests based on this knowledge. In contrast to White-Box tests, the tests still focus on the characteristics of the specification and the outside and do not take concrete code paths into account. [32]

2.1.6 Frameworks

In the following, two frameworks used for implementing tests with page objects are described. The frameworks have been chosen as they are relevant for this thesis and most widely used in practice. There are of course several other frameworks and systems which provide equivalent or more specific functionality.

JUnit

The JUnit Framework [39] is very popular and known for developing automated tests in Java and seen as a de facto standard for it [46]. As described by Sneha and Malle, the "JUnit framework eliminated the gap with developers and testers", as the developer can now test their own implementation against the specification [71]. The currently available version is JUnit 5, which introduced a new architecture, programming and extension model in respect to JUnit 4 [23]. It belongs to the xUnit-family, which provides similar frameworks for different languages [80].

Writing JUnit test classes is similar to the way a general Java class is written. The main difference between them is the way the user has to annotate the class and its methods. Annotations can be added using the @-character before the definition and allow for passing information to the compiler or during the runtime as described in the Java Documentation [2]. The following list describes some of the most used annotations [39]:

- **@Test**
  This Annotation tells JUnit that the method annotated is a test case.
- **@BeforeEach/@AfterEach**
  Methods annotated with one of the given annotations are executed before/after each test case is called. This can, for example, be used to reset some values before/after each test case
- **@BeforeAll/@AfterAll**
  Using one of these annotations on methods results in the execution of the method once before/after the tests are executed. In such methods, one might initialize or destroy objects
- **@Disabled**
  Test cases which should not be executed due to various reasons like wrong assertions can be annotated with @Disabled to ignore them.
- **@Tag**
  When having hundreds of Unit tests, not every test is relevant for specific situations. Using @Tag allows setting tags to test cases and later only execute tests which are relevant.

Test classes are generally structured starting with a variable declaration, followed by annotated methods which should be executed before or after the tests, as described above. Following these setup methods, all test methods follow. While the name of the test methods is not relevant for the execution itself, a good name allows for easily identifing the goal of the test. However, different naming schemes exist and are often adapted to the project's needs.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

To better demonstrate the usage of a JUnit test, listing 2.1 shows a very basic JUnit 5 test, which evaluates if the addition is correctly executed. To demonstrate `@BeforeAll/@AfterAll` the two methods `setUp()` and `tearDown()` are used to set a specific value to the variables.

```
1 import org.junit.Test;
2 import org.junit.jupiter.api.AfterAll;
3 import org.junit.jupiter.api.BeforeAll;
4
5 import static org.junit.Assert.assertEquals;
6
7 public class AdditionTest {
8
9     private Integer number1;
10    private Integer number2;
11
12    @BeforeAll
13    public void setUp(){
14        number1 = 3;
15        number2 = 2;
16    }
17
18    @AfterAll
19    public void tearDown(){
20        number1 = null;
21        number2 = null;
22    }
23
24    @Test
25    public void testAddition(){
26        assertEquals(5, number1 + number2);
27    }
28 }
```

Listing 2.1: Example JUnit testclass (simplified)

One feature not yet discussed but a crucial part in JUnit test can be found in line 26 of the listing 2.1: `assertEquals()`. The method `assertEquals()` is one of several assertion methods used to evaluate the outcome of the call. Each test method consists of three parts: the setup phase in which the environment is set to the expected one (GIVEN), the execution phase where the method gets called (WHEN) and finally the evaluation phase in which the expected outcome is compared with the real outcome (THEN). With the help of assertions these evaluations can be done in a very simple and efficient way, without the need of a high amount of logic in the test case. There are, for example, methods evaluating if a value is true, false or null. The following list gives an overview of the most relevant assertions which are used with JUnit [40]:

- **assertTrue()**: Assert that the given parameter is the boolean true
- **assertFalse()**: Assert that the given parameter is the boolean false
- **assertNull()**: Assert that the given parameter is null
- **assertEquals()**: Assert that both parameters are equal
- **assertArrayEquals()**: Assert that both supplied arrays are equal

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **assertNotEquals()**: Same as assertEquals() but both parameters must not be equal
- **assertAll()**: Assert that all executables do not throw an exception

JUnit 5 also provides assertions for evaluating if a method throws an exception during execution in order to also test negative use cases. Besides the mentioned JUnit assertion-methods, different third party libraries providing assertions like AssertJ and Hamcrest exist, which are recommended by JUnit as well, depending on the required use case.

Besides the mentioned features, JUnit provides a lot of other functionality as well. For example the method *fail()* can be used to force a fail of a test case. However, due to the size of the framework and the rapid development, only the basics can been mentioned in this thesis.

It should also be noted that while JUnit is seen as a de facto standard for testing in Java [46], several other unit-testing frameworks like TestNG [75] exist, which are not relevant for this thesis.

Selenium

The previously mentioned JUnit framework cannot only be used for testing on a component level of a program, but also on a User Interface (UI) one. A popular framework for testing web applications with JUnit is the Selenium Framework [61]. The company defines it as "a suite of tools for automating web browsers". Selenium consists of different projects like the Selenium Webdriver and the Selenium IDE which all share one common approach: "examing the HTML source returned from a web server" [27]. Further Graham and Fewster mention that, while the appearance of a web page still has to be evaluated manually in their case, tests using Selenium would find an HTML element on the web page and compare the expected test with the actual value [27].

While there are also other frameworks by or based on Selenium, like the Selenium Grid [63] which allows scaling the tests developed with the Selenium WebDriver to multiple machines and browsers and therefore reduce the execution time, or Appium [5], which allows testing mobile application user interfaces with Selenium, the following will describe the Selenium WebDriver and Selenium IDE, as they are relevant for this thesis.

The Selenium WebDriver is the key part of the Selenium suite as it is responsible for controlling the web browser as a user would. When referring to the Selenium WebDriver, two components are meant: one being the language bindings and one being the implementation for the individual browsers [61]. The language bindings are provided in different programming languages like Java, Python or C# and allow the developer to create native callbacks to the web browser. The implementation for the individual web browser – also called Driver – is developed and maintained by different parties to execute the language bindings defined by Selenium. The Driver is responsible for calling the WebDriver API of the individual browser, which allows for accessing the web page and executing actions on it. Due to differences of this WebDriver API in different browsers, individual implementations are required. Currently, all major browsers like Chrome, Firefox, Safari or Internet Explorer are supported. Figure 2.5 shows a sequence diagram displaying how a JUnit test is executed using Selenium.

The SeleniumIDE is a browser extension, currently for Firefox and Chrome [64]. There is, however, also a standalone application in development and a command-line tool available for executing the tests. The Selenium IDE provides a graphical way to create and execute test cases or export them to different languages and frameworks like Java and JUnit, using the integrated exporter. The tool also supports logical operations like if/else to develop test cases without even knowing a specific programming language. This tool can be categorized to be a capture & replay tool, as a user can record their behavior on a web page. For more advanced usage it is also possible to define the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

steps manually and to use one of many available commands like *element-click* or *element-exists* to define them.

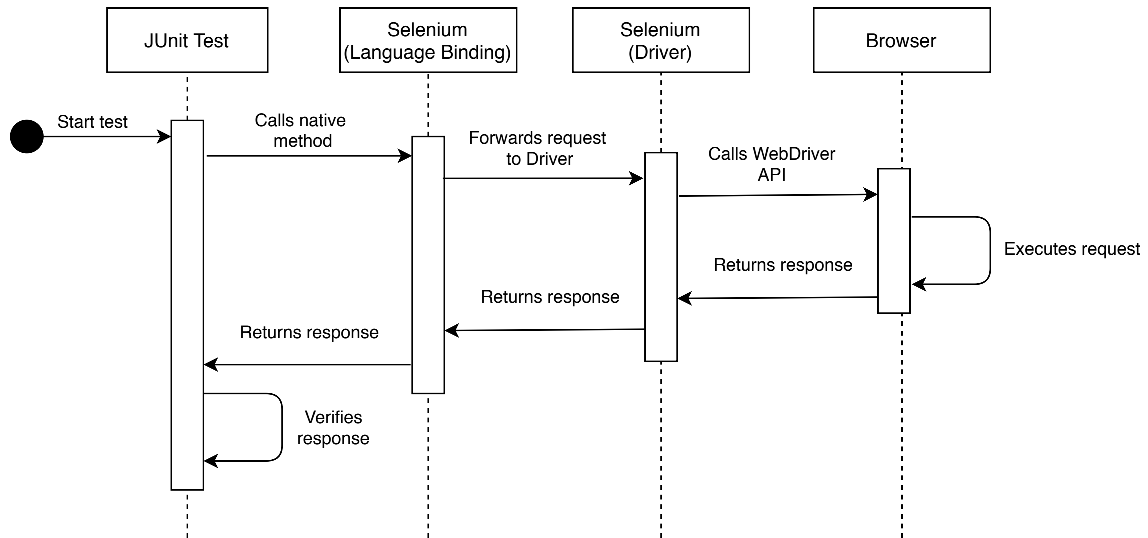


Figure 2.5: Sequencediagram of running a JUnit test with Selenium

2.2 Test automation

As already described before, testing software can take a long time depending on the project and might require a considerable amount of resources. This section focuses on automation of tests and how it can help to achieve better software quality.

The relevance of test automation has been widely discussed in several publications. Dustin, Rashka, and Paul [21] describe automated testing as follows: "The management and performance of test activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool", providing a "significant payback" [21]. In contrast to manual execution of test cases, tests are executed either automatically on the developer's device or in a pipeline on a Continuous Integration (CI) system. It should also be noted that due to the number of tests, especially if run on the developer's device, often only a subset is chosen to be executed. In most cases, pipelines of CI systems are triggered as soon as new commits are pushed to a version control system (VCS), validating the newly written code.

As a general rule, the test pyramid, as described in [16] and figure 2.6, is used to visualize the different layers of automated testing and the proportion of test cases in it. The higher the layer, the fewer tests should be written to get a maintainable test suite. It should also be noted that the complexity and size of the tests increases with each layer due to the broader integration tested. The lower layers, on one hand, have the benefit to run faster, but on the other hand, only test more isolated parts of the system in contrast to higher layers. As illustrated in figure 2.6, most tests shall be written as unit tests. The next layer above contains the integration tests followed by the last layer, the User Interface (UI) tests from which only a few shall exist. The higher the layer can be found in the pyramid, the less frequent the tests are executed. While unit tests are executed multiple times on the developers machine as well as every time the developer pushes their code to the server, UI tests are executed in a periodic task, for example every day at midnight, or before each release. Graham and Fewster also include manual tests in a cloud above the pyramid, showing that manual tests should be included as well, depending on the project's context, but in no ratio to the other automated layers.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

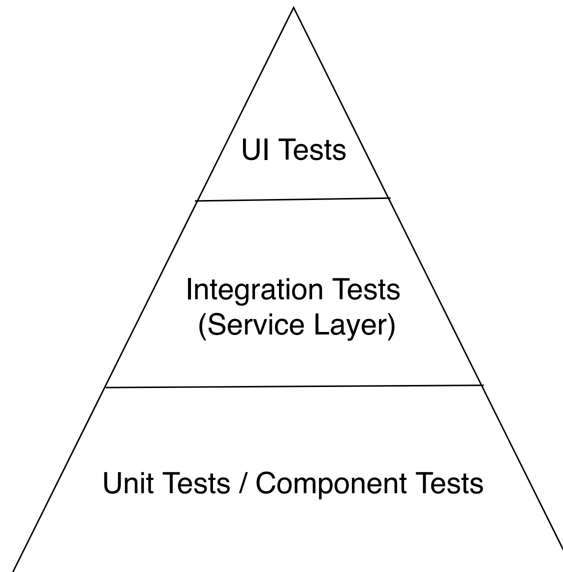


Figure 2.6: The test automation pyramid as described in [16]

Even though test automation aims to reduce the costs of testing, it is not always the case, as Ramler and Wolfmaier describe in [57]. In their publication, the authors provide several calculations on when test automation saves resources like manpower and money, taking into account that the initial setup of automated tests take longer than manual ones. Figure 2.7 shows the calculations from this publication which describes the break-even point of manual and automated testing. V_m are the initial costs of manual testing, V_a are the initial costs of automatic testing. As one can see, the costs of automatic testing are not rising as much as the costs of manual testing do. After n executions of the tests, the break-even point is reached, meaning that costs are saved in the following executions. The authors of [57] describe the number of test executions needed to get to the break-even point as varying between 2 and 20. In [21], Dustin, Rashka, and Paul mention a case study where a reduction of 75% has been achieved through automated testing, with a base of 1750 test executions. Especially the test execution task has had significant speedup of 95% from 466 hours of manual testing to 23 hours of automated testing. Commonly used Continuous Integration systems for testing software automatically are Jenkins [38], GitLab CI [25] or Atlassian Bamboo [8]. Besides the systems executing the tests, different frameworks can be used to define and evaluate them.

2.3 User interface testing

In this section, the process of user interface testing is explained by giving an overview of the fundamentals of user interface testing, their benefits and downsides and the goal. Additionally, different approaches to identifying user interface elements in tests are described. This section ends by describing the page object pattern, a pattern used for model-based testing on which this thesis focuses on.

2.3.1 Fundamentals of user interface testing

As already mentioned before, user interface testing is done in the system test phase. As the user interface is the most visible part of the program for the user it is expected to work well in every environment the user is executing it, hence testing it is indispensable. Testing it frequently includes tasks like verifying if elements are present, if elements are shown depending on the interaction

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

of the user or if error messages are shown in case something in the background failed. This can be either tested again manually by having testers interacting with the application or automatically using various frameworks. While the manual approach cannot be neglected, an automated approach heavily supports the tester by allowing to test different screen sizes and systems. This can be dependent on the type of the application, either different smartphones or different browsers and operating systems for web applications.

While testing the user interface provides good feedback on its quality, there are several downsides to these kinds of tests as well. One of the most relevant problems with user interface testing, which is the base of this thesis's problem, are the volatility issues encountered when doing user interface tests. While user interfaces are very volatile and can change in each release of an application, many of the available solutions for identifying an element require a more static approach. Several options for identifying elements in automatic tests are described in the following subsection. Even in case of small changes in the user interface, like a successor label in front of an input field, many of the access options would either return the wrong element or no element at all, causing some or all of the tests to fail.

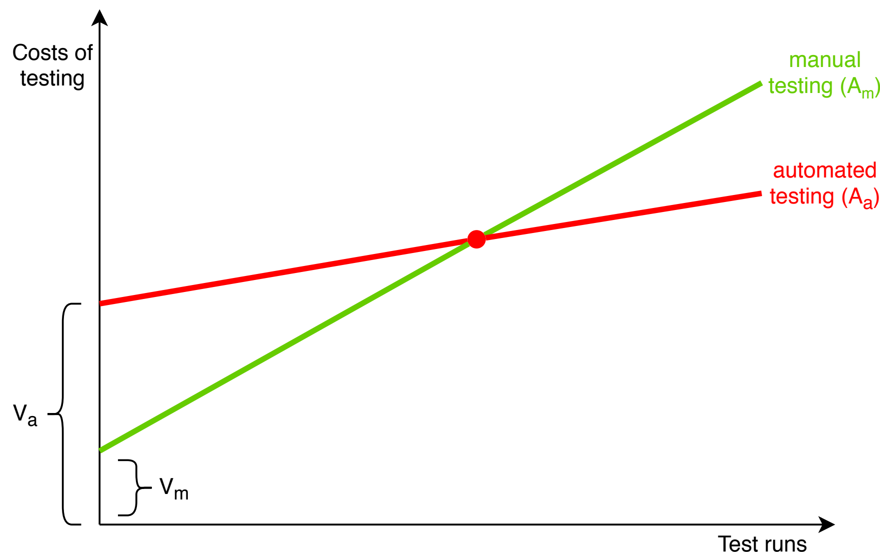


Figure 2.7: Break-Even point of manual and automatic software testing as shown in [57]

2.3.2 User interface element identification

There are different ways how the user interface elements can be accessed by automated tests. As the main focus on this thesis lies on web applications, only identification methods for web applications will be discussed. Before talking about different identification approaches in detail, one common part of all of them will be described: the Document Object Model (DOM) tree.

As a representation of web applications in the browser, the Document Object Model (DOM) is used. The DOM is a W3C standard and described as follows in their specification: "DOM defines a platform-neutral model for events and node trees" [78]. The DOM is often referred to as the DOM tree, because the DOM is structured like a tree as shown in figure 2.8. One can see that the root element acts as a parent for each of its children. Each child can then also act as a parent and have further children. The tree shown is not only a DOM tree but rather an HTML DOM tree, which uses HTML standard tags like `<html>`, `<body>` or `<a>` to describe HTML web pages. Each of them has a special meaning and is expected to include different child elements or attributes. Based on the tree shown in figure 2.8, listing 2.2 shows the textual representation of the HTML file before

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

it was converted into the HTML DOM tree. In this listing the reader can see that each element defines its child between its scope which starts with `<*name*>` and ends with `</*name*>`. While children can be added between those braces, attributes are added directly in the starting declaration of the element as shown in line 9, where an id is set to the paragraph element <p>.

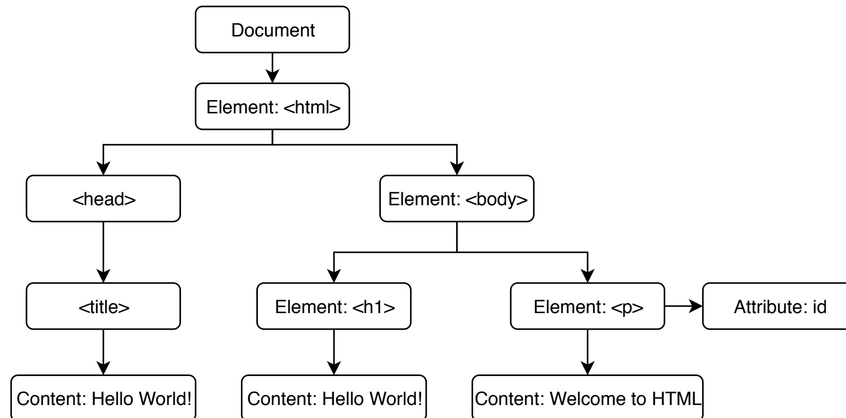


Figure 2.8: The DOM tree displayed as graph

Access via ID

The easiest way to access elements in the DOM tree is by using their ID. Using this approach is safe in case of static web pages where the ID is defined manually, but can also be volatile in case of frequent changes of the IDs – e.g. if they are autogenerated. In this case, the ID might vary with every change. There are also further ways similar to accessing with IDs by providing the class-name or a tag.

```
1  <html>
2
3   <head>
4     <title>Hello World!</title>
5   </head>
6
7   <body>
8     <h1>Hello World!</h1>
9     <p id="textview">Welcome to html</p>
10  </body>
11
12 </html>
```

Listing 2.2: The DOM tree serialized as text

XPath

XPath, which is the abbreviation for XML Path Language, is one of the most widely used ways to access elements when defining page objects and is a W3 standard for accessing parts of XML files [86]. As HTML is akin to XML, the DOM tree of web pages can easily be accessed using XPath. XML files consist of nodes or elements, which are nested in each other and can contain additional information like attributes [33]. "In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes" [79].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

There are several approaches how elements can be accessed by XPath, but they can be split into two categories: Using the absolute path and using the relative path. While the first one, using the absolute path, accesses the element from the root (the topmost node of the XML), the second one, the relative path, uses characteristics of the element.

Absolute paths start with a single slash (/) followed by a sequence of nodes. This approach has the downside that it is very volatile. As soon as one element is wrapped in another element after specifying the path, this XPath is invalid – or at least does not find the right element anymore. `/html[1]/body[1]/div[1]/p[1]` is an example for a XPath expression using absolute path.

In contrast to absolute paths, the relative paths use characteristics of the element. Relative paths start with a double slash (//) followed by a selector. It is also possible to start a path from a relatively accessed element, e.g. to access the second element in a list. The starting path in case of a double slash can now start anywhere in the DOM tree, at the position of the first node specified after the double slash. Table 2.2 gives some examples and explanations for relative paths. The list of possible selectors and their combinations is too long to be included in this thesis, but there are several cheat sheets on the internet, for example [88].

Access via CSS selector

Another approach to access the DOM tree is by using Cascading Style Sheet (CSS) selectors. This approach is similar to the XPath approach mentioned before in many ways, but uses slightly different commands for accessing the DOM tree. CSS selectors are mainly used to style elements on a web page but can also be used to identify elements when executing tests.

The list of selectors and their combinations is again long and therefore not fully mentioned here. There are, however, several lists like [18] available on the internet, which give the reader a better overview. There are 5 basic selectors which are the most notable ones, that select based on the type, class, id or attribute along with the wildcard selector, which selects everything. Combining these 5 selectors already allows for receiving a wide range of elements.

In contrast to XPath, CSS selectors are seen to be more robust towards changes to the web application and also faster in matching elements due to the optimization of the browsers to quickly match CSS files to the web page for styling. Furthermore, CSS selectors are in most cases shorter than XPaths and also easier to read - especially for developers as they are used to style web pages with CSS selectors. However, CSS selectors only allow accessing child elements, while with XPath also the parent can be accessed from the current element.

For better understandability, the aforementioned table 2.2 does not only list and describe relative XPaths but also shows the way CSS selectors would access the same element.

Goal	**XPath**	**CSS Selector**
All elements	//*	*
All *p* (paragraph) elements	//p	p
All child elements	//p/*	p > *
Element by ID	//*[@id='foo']	#foo
Element by class	*Difficult to obtain*	.foo
Element with attribute	//*[@title]	*[title]
First child of all *p*	//p/*[0]	p > *:first-child
All *p* with an *a* (hyperlink) child	//p[a]	*Not possible*
Next element	//p/following-sibling::*[0]	p + *

Table 2.2: Side-by-side comparison of different syntax [87]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.3.3 The page object pattern

The page object pattern is a widely used pattern in model-based testing to gain a better maintainability in user interface tests. It belongs to the in section 2.1.4 described category of test-base models. When testing using the page object pattern, the web page is modeled as an object in the same programming language as used to write test cases [44]. Using best practices, each subpage of the web application is represented as an object and connected with other page objects like the web application. By doing so, "page objects form a layer of abstraction to separate test code from actual elements of web pages under test" [90]. The main benefit of doing so is that only the page object has to be modified when the web page has changed. The tests defined against this page object do not change. Figure 2.9 shows how the page object is connected to the test scripts and the different web pages.

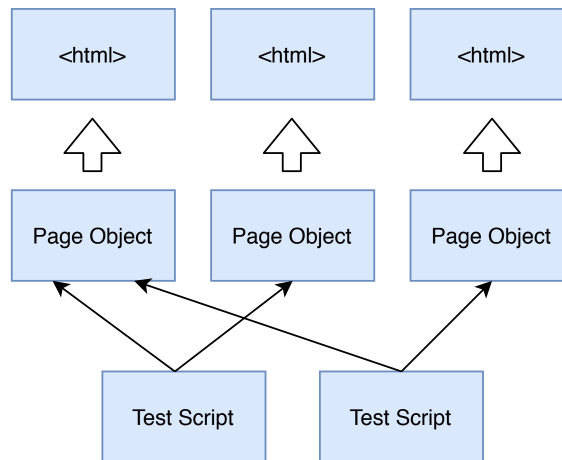


Figure 2.9: Page objects in relation to test scripts

As described in section 2.3.2, there are several ways how elements can be accessed in the DOM tree. In order to build a robust and maintainable page object, it is in most cases more efficient to mix different ways in one page object. For example, one variable might access the DOM tree via a CSS selector while another one uses a XPath. Another relevant best practice for page objects is the already discussed combination of absolute and relative XPaths to receive a robust element identification.

Furthermore, page objects should contain as little logic as possible in order to not influence test scripts in case of updates. Methods of page objects should, following best practices, only reflect the behavior of the web page modeled. This means that methods accessing a paragraph only act as a getter for the text, methods setting data to a text field only set data to that one text field and methods representing clicks on a button should only click the button. In the last case, however, the next page should be returned as a page object as well, allowing the test script to navigate through the web application in the same way a user would.

In the remaining section the page object pattern will be described more deeply using code samples. All code samples are written in Java 8 and use JUnit 5 and Selenium WebDriver in version 3.141.x. The web page used for modeling the page object is shown in figure 2.10 on the left, while the corresponding source code can be seen on the right. For simplicity, this page object should be a simple login form only containing a header text, a text field and a button.

Before discussing the test script which shall evaluate this page, the page object is modeled. Therefore, listing 2.3 shows the code of the page object for the demo page. For demonstration purposes the page object uses three different selectors: one XPath, one CSS selector and one via the id. It should also be noted that, while the page object is modeled using annotations for defining the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Welcome!

Password:

Login

```
1  <html>
2  <head>
3    <title>Demo-Login</title>
4  </head>
5  <body>
6    <h1>Welcome!</h1>
7      <form>
8      <label for="password">Password:</label>
9      <input id="password" type="text" />
10   </form>
11   <button id="login"
12     onclick="window.location='profile.html'">
13     Login
14   </button>
15 </body>
16 </html>
```

Figure 2.10: Web page (left) and its corresponding source code (right)

variables, there is also an inline option available, allowing for accessing the element in the method directly which, again, does not only work with XPaths but also with CSS, id or other described options for accessing elements in the DOM tree. However, the approach using annotation is seen as best-practice, having an overview of all variables used in the page object at the beginning of the file, while the inline option is commonly used for more specific or edge cases like dynamic paths.

The type of the variables in the presented listing 2.3 is a `WebElement` object, provided by Selenium WebDrivers. In the Selenium documentation it is described as follows: "Represents an HTML element. Generally, all interesting operations to do with interacting with a page will be performed through this interface" [66]. While there are other options as well, e.g. defining a `List` of `WebElements` or using the extension of `WebElement` (e.g. `Select`), for simplicity only `WebElement` is used in the demo. The WebElement provides methods which can be executed on the given element, for example `getText()` provides a way to access the text of the element, as shown in line 23 in the listing 2.3. Further relevant methods on a `WebElement` are the `sendKeys()` method for sending actions like text to the `WebElement`, as shown in the listing on line 27, or the `click()` method for executing a click on the element, as shown in line 31 in the listing. The return type of the method, as already described, always represents the outcome of the page. For example, the method `getHeaderText()` returns a string, while the method `clickLogin()` returns the profile page object (not modeled in this example), as the browser navigates to this page after the click. Before the methods described, the page object also defines a constructor for setting up the object with relevant variables. This constructor is called by the PageFactory, a simple factory class for initializing page objects in Selenium, either at the start of a test or when returned by a method (as for example in method `clickOnLogin()`). Furthermore, this constructor also calls `driver.get()` in order to navigate to the page in the web browser. This is needed as we expect this page object as the first page in the test, further pages are then navigated through interactions on the web page.

Finally, a sample test case for evaluating the application will be demonstrated in listing 2.4. The expected goal of the test is to access the page, insert some data and navigate to the profile page. The `setUp()` method, which is executed once before the tests, sets up the Selenium WebDriver by initializing them and defining a timeout. These WebDriver are then closed in the `tearDown()` method after the test is executed. The test starts in line 28, where the demoPageObject variable is initialized via the PageFactory described before. To verify that the correct page is shown, on line 32 the displayed header text is compared with the expected one. In line 33 the password is entered via the page objects method `enterPassword()`. Finally, the login button is clicked by calling the `clickOnLogin()` method in line 34. As described before, this returns the page object of the resulting page after the click, which can be used further to execute methods on it.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class DemoPageObject {

  protected WebDriver driver;

  @FindBy(xpath = "//h1")
  private WebElement header;

  @FindBy(css = "input#password")
  private WebElement passwordField;

  @FindBy(id = "enterButton")
  private WebElement enterButton;

  public DemoPageObject(WebDriver driver) {
    this.driver = driver;
    driver.get("https://peso.inso.tuwien.ac.at/demopage.html");
  }

  public String getHeaderText() {
    return header.getText();
  }

  public void enterPassword(String password) {
    passwordField.sendKeys(password);
  }

  public ProfilePageObject clickOnLogin() {
    enterButton.click();
    return PageFactory.initElements(driver,
    ↪ ProfilePageObject.class);
  }
}
```

Listing 2.3: Example page object

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 import org.junit.jupiter.api.AfterAll;
2 import org.junit.jupiter.api.BeforeAll;
3 import org.junit.jupiter.api.Test;
4 import org.openqa.selenium.WebDriver;
5 import org.openqa.selenium.support.PageFactory;
6
7 import java.util.concurrent.TimeUnit;
8
9 import static org.junit.jupiter.api.Assertions.assertEquals;
10
11 public class DemoTest {
12
13   private static WebDriver webDriver;
14   private DemoPageObject demoPageObject;
15   private ProfilePageObject profilePageObject;
16
17   @BeforeAll
18   public static void setUp() {
19     webDriver = SeleniumWebDriver.getDriver();
20     webDriver.manage().timeouts().implicitlyWait(10,
       ↪ TimeUnit.SECONDS);
21   }
22
23   @AfterAll
24   static void tearDown() {
25     webDriver.close();
26   }
27
28   @Test
29   public void test_enterPassword_Succeeds() {
30     demoPageObject = PageFactory.initElements(webDriver,
       ↪ DemoPageObject.class);
31     assertEquals("Welcome!", demoPageObject.getHeaderText());
32
33     demoPageObject.enterPassword("superSecretPassword");
34     profilePageObject = demoPageObject.clickOnLogin();
35
36     //Do further tasks on the profilePageObject
37   }
38
39 }
```

Listing 2.4: Example JUnit 5 test using page objects

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3 State of the Art

Before defining the requirements on a system addressing the current situation encountered by testers, a literature research was conducted along with a general research of available open-source solutions. This chapter focuses on the results found in the research and describes selected implementations more deeply. Each section in this chapter is dedicated to one implementation – describing its general purpose, advantages and disadvantages as well as its general ability to solve the defined problem.

It should be noted that when searching the internet and especially GitHub, many implementations can be found which are out of support for several years and no longer relevant for this thesis. The last section, section 3.5, will list some of them without giving a deeper insight into their usage.

3.1 APOGEN

APOGEN [73], which stands for Automatic Page Object Generator, was developed over a period of several years and research was done by the project team over the whole period, resulting in multiple publications with an ever enhanced prototype. This discussion will take the latest publication into account. As the source code of the prototype is well described and available on GitHub [4], this section will not only focus on general benefits and drawbacks besides the general description, but also on the implementation part. The main purpose of APOGEN is to generate page objects of web applications including all subpages. By crawling the whole web application, APOGEN can generate page objects not only containing the WebElements for Selenium, but also links between different pages as well as getter and setter for input fields or text displayed on the page. The tool can be downloaded from their GitHub [4] page and can be set up by a technical endued person following their README file. The output files generated by the prototype can be exported to Java page objects.

The team behind APOGEN provides further information on their website [3]. When using the tool, it can simply be started using Maven and configured using the provided *Settings.java* file where the URL, the expected algorithm and further settings can be set. The application then runs by executing the main method of the Java program. Depending on the settings, the whole process is executed, resulting in Java page objects. In case any step needs to be repeated or adapted, the user simply adapts the settings file and starts the process again.

3.1.1 APOGEN's steps

In the following subsection, the different steps taken by APOGEN to create page objects are described in detail. A high-level overview of the tools approach and steps taken can be seen in figure 3.1.

Crawler

In the first step the tool crawls the whole application beginning at a defined starting point and converts the DOM tree to a state-based model which is saved in a JSON file for the next steps. The goal of this step is to get a high-level overview of the application. Besides the graph displaying the connections and pages, this JSON file also contains information about the URL, a list of "click-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

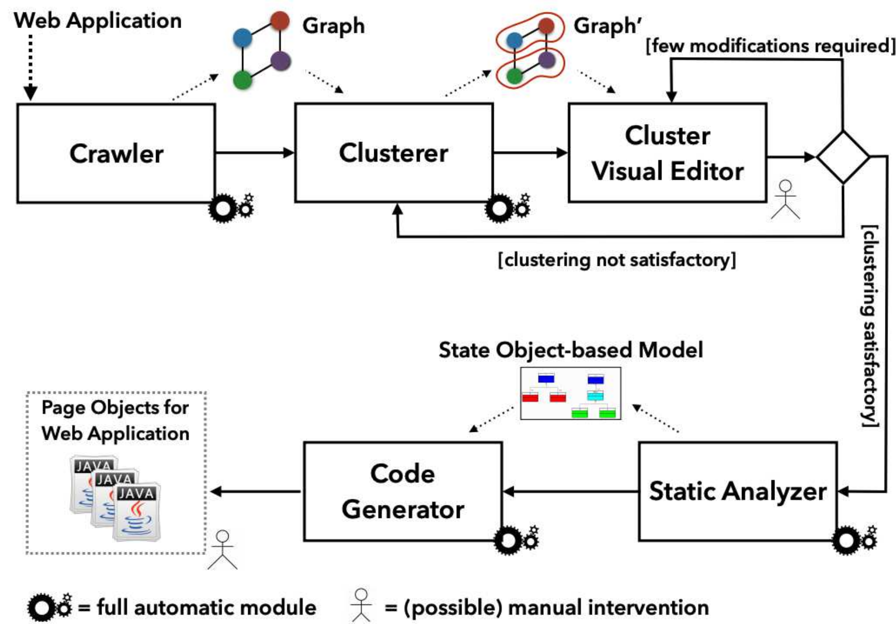


Figure 3.1: High-level overview of APOGEN's approach for web page object creation as shown in [73]

able" elements and other relevant information. As an implementation of the crawler, Crawljax was chosen by the developers. The crawler's default settings were used to decide when to stop the crawling process. Furthermore, relevant input data can be provided such that the crawler could also navigate to pages which, for instance, require a login.

Clusterer

When using the crawled data, the authors experienced two problems. First, duplicate pages are contained in the result multiple times due to different dynamic states a page can have. The second problem was the number of edges between the states, making the visualization of the graph tangled. To overcome these problems, a clustering algorithm was implemented as a second stage. As a result, page objects which could be combined, for instance, because they are the same page but with minor differences, are clustered together. For clustering, two clustering algorithms had been chosen: Hierarchical Agglomerative as of [41] and K-means++ of [6], with the first one beeing the default approach of APOGEN as it was "empirically found to be more effective in producing clusters of web pages close to those manually defined by a human tester" [74]. As a result, another JSON file is created.

Cluster Visual Editor

In the third step in the APOGEN process, a cluster visual editor is used. This cluster visual editor has the purpose to give a human feedback of the clusterer before, as it might result in wrong clusterings. The visual editor is a web-based tool, which reads the JSON file from the previous task and visualizes the clusterings to the user. Three possible outcomes are described by the authors:

1. Clustering is unsatisfactory for the tester. In this case, the clusterer is run again with a different algorithm or different number of clusters.

2. Clustering is satisfactory for the tester. In this case, the tester proceeds to the next step.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3. Clustering is satisfactory, but some changes are expected by the tester. In this case, the visual editor provides capabilities to drag and drop elements and update the JSON file in that way.

Static Analyzer

This process is mentioned to be divided into three parts: DOM diff calculation, FSM modification and merged state object creation. In the first part, the clusters are analyzed and for each cluster a "master" is chosen, representing the base of the page object along with its slaves as sub-elements of the page. This is done by the Static Analyzer through calculating the differences between master and slaves. In the second step, the slaves are combined with the master, taking into account links between the elements of the cluster and to other clusters. This includes some modifications of the edges, as edges might now either point to the element itself or to a different master. The third part then creates a state object-based model of the web application, including a class name, variables and transitions. This model is given to the final stage, the code generator.

Code Generator

The last stage of the APOGEN process is the code generator stage which creates the page objects by conducting a model to text conversion. The JavaParser was used for generating the Java page objects. Each previously mentioned merged master is transformed into one page object containing all required Java class elements like package name, class name and imports along with WebElements including the locators for the element, transaction methods for each transaction between the page objects, actions for data submitting forms and getter for string values on the page.

3.1.2 Tool evaluation

APOGEN approaches the page object generation very well by crawling the whole application and generating all necessary variables and methods for the tester. As a benefit of this approach, the high customization abilities can be seen which expect the program to work in different environments, along with a very good approach for converting the web application to page objects. However, the maintainability approach is not taken into account by APOGEN, as all page objects need to be regenerated at each execution of the tool and existing code or changes are not considered. As the names of variables and methods are also auto-generated, this also means that test cases are required to adapt in certain cases. The biggest downside of this approach, which also stands in contradiction to the expected behavior of a maintainable approach, is the regeneration of an immense number of variables and methods which are not all required by the tester. As the tool is not aware of the test cases and therefore not aware of the used variables and methods a lot of dead code is regenerated on every execution of the tool which also needs to be maintained by the tester. All the mentioned problems are in contradiction to the expected behavior of model-based testing and a maintainable approach to it.

3.2 Page Modeller

The Page Modeller is a browser extension supporting major web browsers like Chromium-based browsers, Firefox and Opera. The extensions code can be found on GitHub [53]. It is an actively maintained tool in version 1.5.4 at the time of writing. As of this version, the tool supports code generation for Java, C#, the Robot Framework and Puppeteer.

The tool is installed via the extension store of the browser and can then be used in the developer console. It is opened next to the page which is expected to be modeled and the target language selected. After that, elements can be added to the page object by clicking on the *select* button

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

and clicking on an element on the webpage, which adds the element to the Page Modeller. In the window of the Page Modeller, every added element can be changed regarding the name and chosen locator. Furthermore, the user can show the selected element on the webpage. After all elements have been selected, the user can export the code by copying the generated code to existing or new page objects.

A big benefit of this tool is the easy-to-use approach, allowing everyone with a supported browser to easily install the tool and generate WebElements for page objects automatically. The tool also does not require the user to know about XPaths or any identification of WebElements as the user can simply click on the element and gets it converted to a WebElement. As another benefit of this selective approach the reduced number of variables can be seen, only including those really required by the tester. However, this approach also has some downsides. Pages, for instance, need to be modeled each time a page object needs to be generated – or at least those parts which should be generated. Furthermore, the tool only provides the generated code segment in the browser, which then needs to be copied by the user and pasted in the either newly generated or existing page object file in the project. This again does not provide a lot of benefit in contrast to manually finding the paths and entering them to the code manually, as most XPath support tools also allow the user to simply select an element and get the correct path for it instantly. Additionally the tool does not provide information on which paths are invalid or need to be adapted by the user. While this approach allows for reusing existing page objects and only adapting required paths, which enhances the maintainability aspect of page objects, a lot of manual interaction by the user is required to do so.

3.3 Selenium Page Object Generator

When searching on the Chrome Web Store for extensions, the Selenium Page Object Generator [67] can be found, which also has a GitHub page providing insides to the code [68]. It is marked to be in a beta state currently, however, the extension did not get an update on the Chrome web store or a commit on the repository since early 2018. The tool currently supports an export to Java, C# and the Robot framework.

For generating a page object, the user navigates to the desired page in their browser and opens the extension. After that, the target language can be chosen along with the page object name and an optional destination object name representing a page object where the user gets redirected when clicking on the page. The generated page object looks as expected, providing constructors, each WebElement as a variable and convenience methods for each element, like method to execute a click on WebElements representing links. Each method is provided with documentation.

A big advantage of the tool is the easy to use chrome extension which allows even non-developers to create page objects easily. Another benefit of the tool is the multi-language support along with the possibility to adapt the template of the exported page objects in the options. By doing so, individual preferences can be taken into account when exporting the page object. A downside of this approach is again that page objects need to be generated all at once, potentially requiring adaptions of the methods every time generating the file. The provided files also contain each element displayed on the page along with methods for them which can be a big boilerplate if not needed by the tester. A big disadvantage for medium- and large-sized web applications is that in order to generate page objects, each sub-page of the web application needs to be accessed manually and the generate-task executed by a tester. This not only requires a lot of time to do so, but also requires the exported file to be adapted to the projects needs. Furthermore, each generation of a page object regenerates the whole file again. Therefore the generator is expected to be used only for applications with a small number of pages. For bigger web applications the generation process

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 3.2: Screenshot of Selenium Page Object Generator

time is reduced, but a considerable amount of additional time is required in order to adapt the page objects to the projects needs.

3.4 SWET - Selenium WebDriver Elementor Toolkit

The Selenium WebDriver Elementor Toolkit, which can be found on GitHub [70], is a tool for generating page objects. At the time of writing, the repository is actively maintained, with the last commit only days before the time of writing. The tool is said to be running on all major platforms like Windows, Mac or Linux. As a template engine Jtwig is used for generating Java page objects, which is said to support PHP Twig as well [70].

To generate page objects, the user first launches a browser window from the tool and navigates to the website which should be tested. Then, the user injects a search script to the opened web page through the tool. After that, the user can simply click any element on the page while holding the control button (CTRL) to add it to the page object. When doing so, the injected script is triggered which allows the user to define a variable name and select the way the element should be accessed. In the end, the code can be generated by clicking on the corresponding button in the tool, allowing the user to copy the code to new or existing page objects.

Using this tool for page object maintenance provides the benefit that only variables are defined which are required in the page object. Existing page objects are adapted by the user manually, leaving tests and other dependencies working as well as only relevant parts of page objects changed. On the other hand, each variable has to be reselected if a page object needs to be adapted or regenerated. Furthermore, as only parts of page objects are generated, no real page object generation is done, leaving the user with as much or more work on maintaining page objects as done manually.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.5 Further findings

During research, several other tools have been found in addition to the ones described above. Three of them will be described shortly in this section. However, these tools are not seen to be relevant when defining requirements, as they are either unmaintained for several years or can only be used with a lot of additional effort.

3.5.1 WTF PageObject Utility Chrome Extension

Besides different tools usable in combination with Selenium, the Web Test Framework (WTF) in their GitHub repository [85] also provides a Chrome extension which can be used to automatically generate page objects. The extension must be installed manually through the developer options of the chrome extensions instead of using the usual way of the Chrome store. Furthermore, the project's last commit was in November 2014 and the extension is not working with the current versions of Chrome, resulting in an error when trying to execute page object generation.

3.5.2 PageObject-IO

The page object generator from pageObject.io [54] can also generate page objects based on HTML templates of Angular. However, the website only mentioned Angular 1 and 2 explicitly whereas as of the time of writing version 9 of Angular is used in production, which might result in wrong page objects. The last commit on GitHub for this project was in 2017. Furthermore, the user has to manually copy and paste the code to their web page in order to generate a page objects' code which then again needs to be copied back to the project. This results in an immense workload even for smaller projects. Lastly, Vue.js and React – two commonly used frameworks in development – are not supported by the tool but mentioned to be available soon.

3.5.3 Selenium Code Generator

The Selenium Code Generator [62] can be used to generate page objects, methods and links automatically. For generating the page objects, not the page itself is used but rather elements provided in a configuration file. While this allows the user to maintain all XPaths in one place, it requires manual interaction to get the XPaths and to find invalid ones, once the page object needs to be updated. Furthermore, using this approach, the configuration file also needs to be maintained in addition to the page objects.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4 Motivation and requirements

This chapter focuses on the requirements for an approach that enhances the maintenance of page objects in the development life cycle. Before the requirements are defined, quality characteristics like maintainability and traceability and their relevance for the testing life cycle are described in section 4.1, along with a discussion of problems current implementations face in contrast to the intended behavior. Based on this, section 4.2 will then define the requirements for an approach enhancing the current situation. These requirements are defined based on features of existing tools as well as best practices in order to enhance the current situation in the field of maintainability.

4.1 Quality characteristics and their relevance for page objects

Non-functional requirements are defined in software projects to increase quality aspects of the software [43]. The ISO 9126 standard defines six quality characteristics as non-functional requirements: functionality, reliability, usability, efficiency, maintainability and portability [36]. As page objects need to be adapted once the user interface changes, maintainability can be seen as the most relevant quality characteristic for them. The IEEE standard glossary of software engineering terminology (610.12-1990) [34] defines the term maintainability as follows: "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment". Further characteristics belonging to maintainability according to [34] are extendability - how easily "a system or component can be modified to increase its [...] functional capacity" - and flexibility - how easily "a system or component can be modified for use in applications or environments other than those for which it was specifically designed". Wallerstorfer also mentions that "maintainability indicates how complicated it is to modify the code to

- correct faults,
- improve quality attributes (e.g. performance),
- add functionality,
- and conform to changing requirements. " [81]

Due to the often and reappearing changes to the user interface, page objects need to be updated very frequently - depending on the quantity of the changes. Hence a good maintainability and extendability is required in the software testing process to reduce the resources required for it.

Besides quality characteristics focusing on the maintenance of the page objects, there is also the non-functional requirement of tracability which is relevant in software testing, especially when using page objects. Traceability is defined in the mentioned IEEE document as follows: "The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another" [34]. When using page objects, traceability comes into effect when version control systems (VCS), like Git, are used in development for them. They allow for easily tracing changes made in the project on file-level in a very granular way. VCS allow for seeing which lines of code have been changed, providing a reviewer an overview of changed paths and variables in case of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

page objects. However, good and understandable traceability is often not available due to unnecessary changes in the files, especially in case of automatically generated files. In this case the whole file is marked as changed on each regeneration, rendering it useless for a reviewer.

Another non-functional requirement, which is mentioned in the requirements below, but not directly relevant for software testing, is usability. Even though not needed for page objects in general, certain requirements also enhance the usability of the page object maintenance process, therefore a definition should be mentioned here. Usability is described in [34] as follows "The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component".

4.2 Requirements

In the following, requirements for an approach shall be defined, which ought to solve the problems testers encounter when using page objects in context of maintaining them as described in chapter 1. Each requirement was either taken fully or partly from existing state of the art projects as described in chapter 3 or newly defined based on best practices in software development or software testing. For better understanding of each requirement, the order of the requirement list does not reflect whether they are newly created, adapted or taken fully taken from existing projects. Each requirement describes their origin and figure 4.1 in the end of this chapter summarizes them for a better overview.

- **R1: Reuse of existing page objects**

  As a key feature of maintainability, existing page objects shall be reused as much as possible. Reuse of existing page objects means that characteristics belonging to the object like the package name, the imports, other classes or interfaces the object extends or methods are preserved during the whole process and do not change afterwards. This prevents a lot of manual interaction needed to bring the object back into the existing project and further results in a better traceability of changes.

  Reuse of page objects also includes importing and exporting in the desired programming language. For doing so, functionality in parsing objects is needed as well as functionality for writing objects in the chosen programming language. As the focus lies in the maintenance of page objects, it is sufficient to export the page object in the same programming language as it was imported. However, language-specific characteristics need to be taken into account, like using the correct syntax in order to minimize the effort needed after replacing the old page object in the project.

  This requirement focuses on the maintainability and traceability of the page objects' code and is newly defined, as no tool found currently supports the import or export of page objects.

- **R2: Maintaining only relevant page objects**

  In order to preserve a good overview of the git history and the changes of each commit, only a selected subset of page objects shall be maintained and generated. Therefore, it should be possible to select the desired page objects manually and check, and potentially, update only the selected ones.

  This requirement on the one hand enhances the usability for users, as they only have to deal with a subset of page objects and further enhances the traceability, as only page objects with

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

changes are added or modified in the VCS. This requirement implicitly exists in some tools, which allow for copying and pasting the generated code to the file, hence only maintaining page objects which really change. However, as non of the tools provide the functionality of R1, the approach needs to take care of the requirement to only maintain a subset of page objects.

- **R3: Overview of multiple page objects**

  When maintaining page objects, the user shall also be able to handle multiple page objects at the same time. By doing so, the user benefits from a better overview on the maintained page objects and further gets the opportunity to easily switch between them to think about possible enhancements and rearrangements of variables. This also allows changes after one page object has been exported and does not come with the overhead of starting over again once proceeding to another one.

  Current tools do not support this kind of overview. Discussed tools either do not provide an overview at all or just for one page object at a time. Once the user proceeds to another page object, one has to start all over again when the previous page objects needs additional changes. Therefore this requirement was newly defined to better support the user with their page object maintenance. The user benefits from a better usability when provided with this kind of overview, which also allows for reduction of the time spent maintaining page objects.

- **R4: Maintaining only relevant variables**

  Similar to requirement R2 described before, an adaption of a page object shall only change the code of the adapted variables. This subset includes those variables which are changed by the user, preserving all unchanged variable data. For instance, if the user only changes the path of the variable, the variable type and name should not be changed. In conjunction with requirement R2, this means that a change on the page object only changes the necessary parts of the code. This also allows external users to review the required code changes without a lot of unnecessary code changes around them, enhancing the tracability of the changes and hence a better traceable VCS history. Another benefit of maintaining only the relevant variables is that no boilerplate or even dead code needs to be maintained by developers who do not know which variables are really required by tests.

  This functionality of the requirement can already be found in other implementations, which allow maintaining only a subset of variables on the page. Similar to R2 this requirement mainly focuses on the traceabilty aspect of page objects.

- **R5: DOM identification**

  The most important part of page objects are the access paths of the variables. A key feature for enhancing the maintenance of page objects therefore is their simple selection on the DOM. In order to support the developer, it should be possible to automatically define these paths after user interaction without requiring the user to define them manually. This shall be achieved by simply clicking on the element for which the path shall be added to the variable. Providing visual support for the selected element can be seen as a sub-requirement. However, it should still be possible to manually adapt these paths if needed.

  There are tools which already support this behavior described in chapter 3. As this approach is seen to be the easiest way for detecting and adding the path to a variable, this requirement

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

allows for reducing a considerable amount of time needed to find correct and robust paths and hence eases the maintainability of page objects.

- **R6: Adding and modifying variables of page objects**

  While previous requirements R1, R2 and R4 focus on the general maintenance of the page objects and their variables, this requirement lies its focus on the general CRUD (Create, Update and Delete) methods of the variables. It should be possible for the user to add new variables to the page object, edit and also delete them. While the name shall only be changed by user manually, the path should be added or modified as described in R5.

  This requirement can be found in some tools as well but was extended to support more actions on the given variable. This requirement also tries to reduces the time a tester has to spend on the maintenance process by providing a better interface for changing characteristics of a variable.

- **R7: Overview of variables and their paths' validity**

  In order to give the user an overview of all variables on the chosen page object, the user shall be provided with a listing with all variables and their parameters. Furthermore, the validity of each variable's path shall be verified after user interaction and their status displayed. This allows the user to easily detect invalid variables and change the path accordingly.

  Some of the previously mentioned tools also include this functionality, based on which this requirement is chosen. This requirement reduces the time needed for a tester to verify multiple paths of the page object and therefore accelerates the maintenance process.

- **R8: Display page object variables on page**

  While the previous requirement gives an overview of the validity of each single field, the user shall also gain an overview of the elements which are covered on the web page in order to determine if all expected elements are covered, or if there are any new elements which need to be taken into account and added to the page object. Therefore, the user should be able to select either one or multiple variables of the page object and visually see the element on the web page.

  This requirement is currently only supported by one tool, which also allows for displaying the chosen element on the page as described in section 3.2. However, this requirement extends it by additionally allowing the user to display all available paths of the defined variables on the page object. When using a feature based on this requirement the user saves a considerable amount of time when analyzing their variables as they can directly see where the element is located or which elements are covered by the page object.

RQ1	RQ2	RQ3	RQ4	RQ5	RQ6	RQ7	RQ8
NEW	ADAPTED	NEW	EXISTING	EXISTING	ADAPTED	EXISTING	ADAPTED

Figure 4.1: Overview of the requirements' origin

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5 Prototype concept

The following chapter will discuss the conception phase of the prototype developed to evaluate the requirements defined in chapter 4. Section 5.1 will discuss the process of finding the most suitable application type, further characteristics for the prototype and will present the result of it. Based on the requirements, section 5.2 will then present the planned user interface as mock-ups along with the expected usage behavior and its features.

5.1 Prototype characteristics

While a list of specific requirements for an application used for page object maintenance has been defined, several characteristics of an application are free to be chosen when implementing it. Characteristics not yet defined are for example the application type – where and how the application is executed –, the programming language the tool is written in or the programming languages the tool supports. This also includes external frameworks which shall be supported in the exported file.

5.1.1 Application type

The most important question when starting an application in this context is the selection of the application type. A wide range of possible types is available, however, not all are suitable for the development. For instance, a command-line interface (CLI) tool can not, due to the missing user interface, be seen as a possible type for the application. The following three options have been chosen to be the most appropriate for a prototype on which the evaluation shall be conducted and will be discussed more deeply in their advantages and disadvantages:

Browser extension

Most of the discussed state of the art tools are provided via a browser extension. One big advantage of browser extensions is the close proximity to the browser itself, consequently the easy access to the DOM tree and its elements. Furthermore, the foundation of the application is already laid by the browser, only requiring the developer to develop the application for the browser and gain broad support of devices and operating systems.

Downsides on the chosen approach are on the other hand the more complex interaction to and from the application regarding the file-system. As the browser acts as a layer between the system and the extension, for example, file modifications have to be done via a download and then manually handled by the user. Furthermore, this approach heavily depends on the functionalities of the browser and the options provided.

Stand-alone application

A stand-alone application in contrast to the browser extensions allows much more access to system functionalities. Local files could be changed easily via this application type as the tool would have access to the element. Modifying the file using this approach is most certainly the most convenient way for the end-user as in best case, no interaction is needed for it. Another benefit for the developer is the free choice of programming language the application shall be written in.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

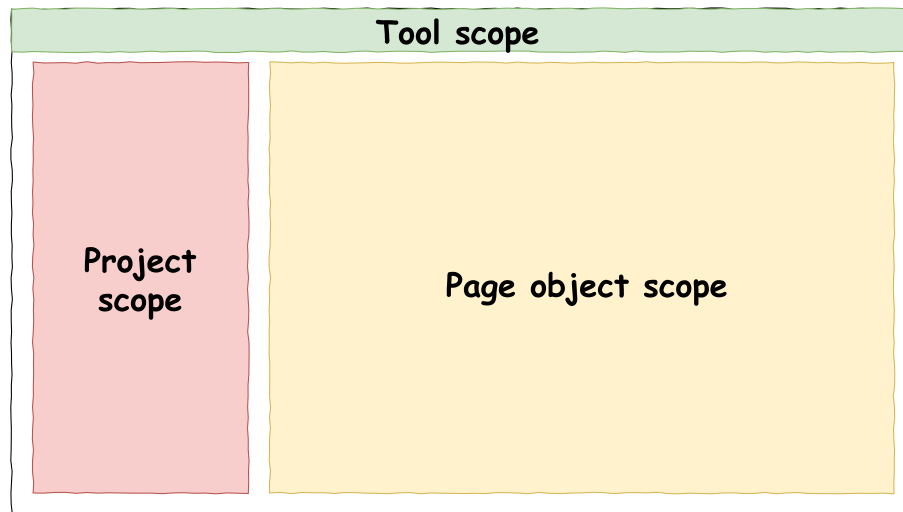


Figure 5.1: Basic user interface parts

However, there are also downsides using this approach. The main downside is the missing connection to the browser and its APIs. The tool would have to provide a connection to the browser, which can be different in different browsers and operating systems. Operating systems can also pose a problem as different operating systems could handle same things differently and might require an implementation for each of them.

IDE plugin

Another method for implementing the prototype would be to build it as an Integrated Development Environment (IDE) plugin. By doing so, the user would have the toolset directly in the same environment as the tests and page objects, which would result in the best user behavior possible. Furthermore, the IDE would provide further functionality supporting the development process, which could also ease the creation of page objects. Additionally, the plugin would work on every system the developer uses the IDE on, leaving nobody without support.

There are, again, also some downsides on this approach. First, the developed prototype would have a lock on the vendor the plugin is developed for. The main parts of the prototype could be adapted to different IDEs, however, this requires additional effort. Second, the programming language would be given due to the required language for plugins depending on the IDE. Third, like the stand-alone application, this approach would have to find a way to communicate with a browser to gain access to the DOM tree and the functionality of the page, which requires additional effort.

Based on these advantages and disadvantages, the decision fell to a browser extension due to the easy connection to a web browser, which is an essential part of a tool like this. The browser connection from a stand-alone application or an IDE plugin might also be possible, however, most likely harder to accomplish and harder to fit in the time frame of this thesis. Moving to an IDE plugin can be seen as future work, as the advantages regarding code modification directly in the IDE are also convincing. Furthermore, it should be noted that the focus of a browser extension was put on building an extension for Chrome as the market share of it was at about 68 percent in April 2020 [19].

Choosing a browser extension as the way to go, this also answered the question regarding the programming language the tool shall be written in, as browser extensions are commonly written in JavaScript.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 5.2: Mock-up of the toolbar on top of the application

The last decision has been taken regarding programming languages the prototype shall support. As the prototype shall only reflect the general use of a tool with the given requirements, a single programming language was decided to be sufficient. The chosen programming language for the process of page object import and export was Java, as Java is, at the time of writing, the most used one according to TIOBE [76]. Furthermore, as the exported page objects also depend on external frameworks as well, Selenium and their WebDriver has been chosen as a requirement of the project when using the prototype. This is relevant for the correct internal behavior of the generated page object – e.g. import statements and objects used for the variables.

5.2 Expected usage behavior and mock-ups

Using the list of requirements from section 4.2 and taking the decisions of section 5.1 into account, mock-ups were created based on which the user interface shall be implemented. Furthermore, the expected usage behavior will be shortly described as considered while designing it.

When designing the user interface of the prototype it was decided that the basic structure of an IDE user interface shall also exist in the tool. Therefore, it was decided to build a single page application and split this window into three parts as it can be seen in figure 5.1. The top part (green) is expected to be a tool specific area where general actions are located. The left element (red) reflects project-related actions whereas the right element (yellow) reflects page object-related actions. For better feedback during the usage of the application, pop-ups have been chosen as a companion to the single page application.

In the following, each part of the split window will be explained further:

5.2.1 Toolbar

The top area can also be seen as a general toolbar to the application and contains tool-specific functionality. The main content of this toolbar are three buttons:

- ***Open browser* button:** In order to fulfill requirements which target the web page, like displaying an element on the page, a browser window is required. By clicking on this button a new browser window should be opened with which the tool can interact with for selecting elements or evaluating their existence.
- ***Load page object* button:** This button is required for the import requirement. When clicking on this button, the file chooser of the browser shall be opened such that the user can select a page object file to be used. This file then should be imported and processed by the tool to display its content.
- ***Download page object* button:** For fulfilling the requirement of downloading only relevant page objects, this button shall provide a download chooser provided by the browser for exporting page objects in the desired programming language. As the project can consist of multiple page objects, the export shall be done via a zip-archive.

The toolbar as described can also be seen in figure 5.2, which shows how the toolbar is expected to look like in the prototype.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

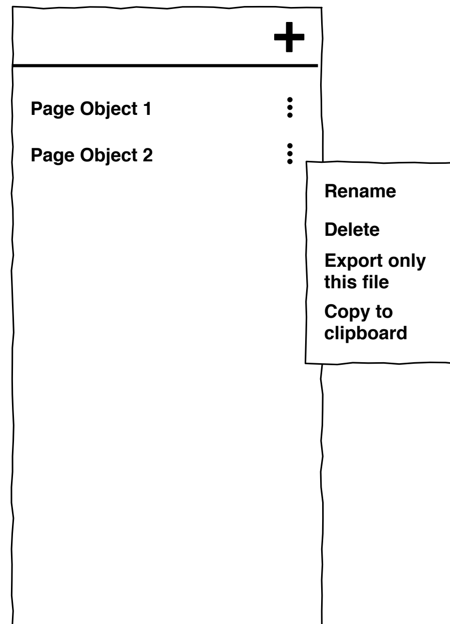


Figure 5.3: Mock-up of the project-related view on the left of the application with an opened drop-down menu for further actions

5.2.2 Project area

The left area of the application is dedicated to the project the user is currently working on. It mainly contains a list of all page objects currently processed by the tool and allows for performing actions on them. Page objects opened and imported via the import option will be added to that list as well as page objects which are created by using the button in the small toolbar above the list.

Actions which can be performed on the page object can be accessed through the three dots on the right side of the entry and are planned to be the following:

- **Rename:** Using this option the user shall be able to rename the page object.
- **Delete:** By selecting this option, the page object is removed from the project and is no longer included in the project list or export.
- **Download only this file:** As the download option in the toolbar exports all page objects at once, this option shall allow the user to download only the single desired page object.
- **Copy to clipboard:** This option provides the user with even more convenience by copying the page object to the clipboard instead of writing it to a file. While both download options require the user to navigate to the file in the file-system and then move it to the correct position, this option allows a fast and easy interaction with the opened IDE and the files there. Furthermore, some IDEs, like IntelliJ, provide the option to compare the file with the clipboard, allowing the user to easily identify changes and merge the changes with the existing file [35].

The expected view for the project area as a mock-up can be found in figure 5.3. By clicking on a row in the provided list, the page object area shall be updated to reflect the chosen page object and its content.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

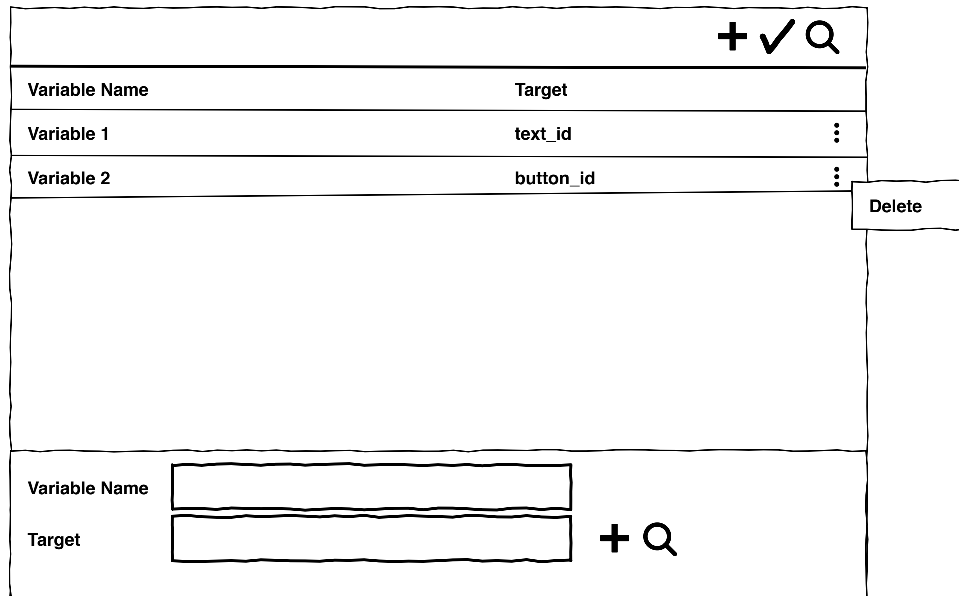


Figure 5.4: Mock-up of the page object view allowing modification and verification with an opened drop-down menu for further actions

5.2.3 Page object area

The right area is the main interaction view for the user, as here actions on the page object itself are performed. Therefore this part can also be seen as the page object area. This view again is split into three parts to keep different information and actions on the page object separated. The top of the page object area is a toolbar where actions on the chosen page object can be executed. The middle part of the area provides a table view of all variables which can be found on the page object. The bottom part allows modifying the values of the selected element in the table.

In the toolbar, the following actions can be executed using the provided buttons. Each action shall be executed in the browser which was opened and provided by the tool.

- **Add new variable:** This action allows adding a new variable to the selected page object. When clicking this option, the user shall be able to select an element on the web page currently displayed. After clicking on the element the variable shall be added to the list using the provided target and a random variable name.
- **Verify page object:** By clicking on this button, the displayed page object shall be verified against the web page displayed in the browser. After verifying each variable, the row shall either be green for elements found or red for elements not found.
- **View all elements:** This button highlights all elements of the page object which can be found in the DOM tree of the current web page.

The table view in the middle of the area lists all variables of the page object. Each row also provides the option to access a menu using the dotted button on the right. The function which can be accessed through this menu is the following:

- **Delete variable:** When clicking on this option, the variable is deleted from the page object and no longer considered for validation or export.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

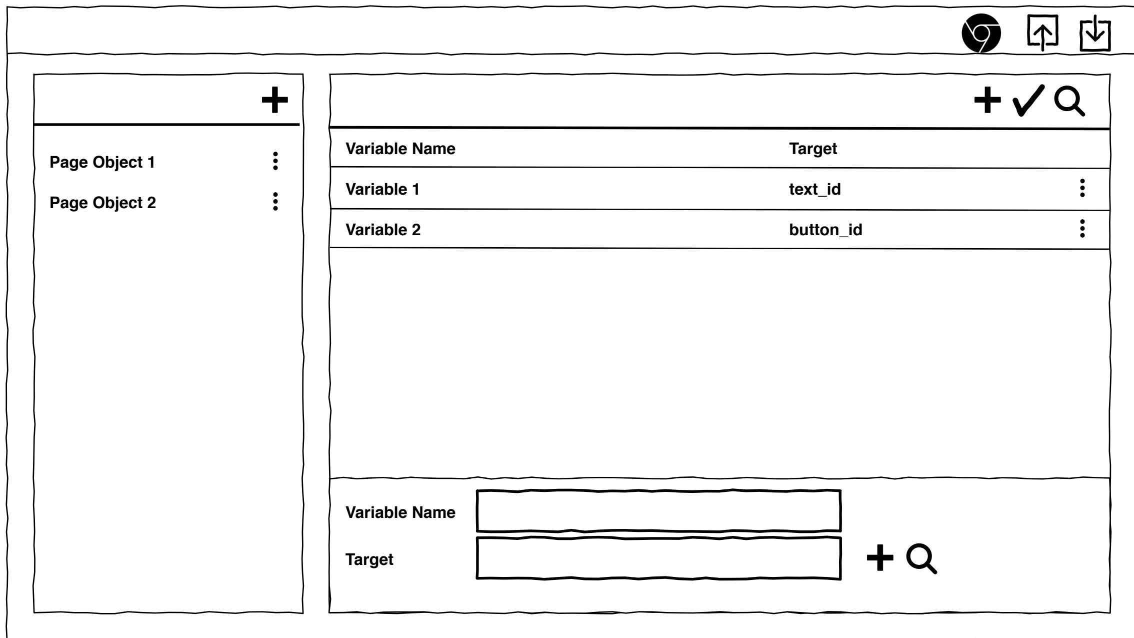


Figure 5.5: The full user interface as planned

When selecting a variable on the table, the lowest part of this area should contain the information of it and allow modification of the data. This shall be done by providing two text fields for editing the variable name and target for the page. Furthermore, two supporting actions are provided for the target, one being the *Select target in page* option allowing for simply clicking on an element on the page to receive the target, and the second option *Show variable on page* for displaying this single variable on the page. Both actions shall again be executed using the browser window opened through the tool.

The planned user interface of this area can be found in figure 5.4, displaying all three parts of the page object area in the mock-up.

Based on the discussed parts, the full user interface as mock-up can be seen in figure 5.5. It should be noted that these mock-ups only represent the basic concepts of the user interface as planned to be implemented. The user interface of the implemented tool can be found in the following chapter, which discusses the implementation process and provides screenshots of the application used for evaluation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6 Implementation

This chapter will focus on the implementation of a prototype called POGito, which is used for the evaluation of the requirements defined in chapter 4. In contrast to the concept discussed in chapter 5, this chapter has a more technical focus and provides code samples for better understanding. Section 6.1 will provide basic knowledge on how a Google Chrome extension is built and provides background on two external frameworks used in development. Section 6.2 will then focus on the prototype implementation itself, providing general information as well as information on the internal structure, screenshots and insights into chosen code parts describing how features work.

6.1 Implementation basics

In this section, the basics of the developed prototype will be explained. Subsection 6.1.1 describes how a Chrome extension is built and discusses several special characteristics along with some relevant code samples. In order to easily use the extension in both Chromium-based browsers and Firefox, a framework called WebExtensions API Polyfill [83] was used, which will be further described in subsection 6.1.2. The last subsection, subsection 6.1.3, will then extend the discussion on Selenium and their Web-Driver framework started in section 2.1.6 to its usage in the extension for an easy to use connection to the browser.

6.1.1 Chrome extension basics

In the following, the basic structure of a Chrome extension is discussed. It should be noted that while the discussed topic might also be valid for other Chromium-based browsers, and even to a certain extend for other browsers, the discussion is made only for the Chrome browser by Google due to the high popularity with a market share of about 68 percent as of April 2020 [19]. The following subsection will discuss precautions made to support more browsers.

The base of every Chrome extension is a manifest file in a JSON file-format. This file contains basic mandatory information like the name, the version of the extension and the version of the manifest - which at the time of writing is version 2. Furthermore, additional extension specific information can be found in there like the extension's icon, permissions the extension requires or scripts which shall be executed. These scripts are written in JavaScript and can either be background scripts or content scripts. Listing 6.1 shows a shortened version of POGito's manifest file.

As the user needs to interact with the Chrome extension, a user interface is also required. These can either be small pop-ups, like in most of the browser extensions, allowing the user to trigger a small set of actions, or bigger ones like POGito which provide a rich user interface experience like a native application. Pop-ups can be declared in the manifest directly using the `page_action` entry – which is not shown in the example because it was not used in POGito –, while windows need to be declared in the background scripts and opened from there. The differences between background scripts and their counterpart content scripts are the following:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
1 { "description": "POGito helps you creating and maintaining your
  ↪ page objects",
2   "manifest_version": 2,
3   "name": "POGito",
4   "version": "1.0",
5   ...
6   "icons": {
7     "16": "icons/icon16.png",
8     ...
9   },
10  "browser_action": {
11    "default_icon": {
12        "16": "icons/icon_menu16.png",
13        ...
14    },
15    "default_title": "POGito"
16  },
17  "permissions": [
18    "tabs",
19    ...
20  ],
21    ...
22  "content_scripts": [
23    {
24      "matches": [
25        "<all_urls>"
26      ],
27      "js": [
28        ...
29        "assets/record.js"
30      ],
31        ...
32    }
33  ],
34  "background": {
35    "scripts": [
36      "assets/background.js"
37    ]
38  }
39 }
```

Listing 6.1: Excerpt of POGito's manifest file

Background scripts

Background scripts provide the connection to and from the browser. Like all scripts used in the extension, background scripts need to be declared in the manifest file. JavaScript files declared as a background script can then trigger actions to the browser or provide endpoints for listeners, which are notified by the browser in case a specific event happens. The code fragment in listing 6.2 consists of both a receiving and a sending part and was taken out of the tutorial for Chrome extensions [24]. This code gets triggered by Chrome when the extension is installed on the browser. It then saves the color green to the storage of Chrome and displays the log "The color is green" in the console. The variable `chrome` is provided by the Chrome browser itself, which allows for accessing all browser APIs like runtime or storage. This variable is globally available to all scripts

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

running in the Chrome extension by default. An overview of all possible Chrome APIs can be found online [12].

```
1 chrome.runtime.onInstalled.addListener(function() {
2    chrome.storage.sync.set({color: '#3aa757'}, function() {
3       console.log("The color is green.");
4    });
5 });
```

Listing 6.2: Sample background script method [24]

Content scripts

In contrast to background scripts, content scripts are run in the context of the web page, allowing to read and modify the DOM [17]. As content scripts are run in a different context than background scripts, they can not be called directly from a background script or vice versa. Furthermore, only a small subset of Chrome APIs – `i18n`, `storage` and a subset of `runtime` – can be used by content scripts directly. All other communication to the extension needs to be handled through messages. A simple message conversation is shown in listing 6.3, which was also taken from Chrome's tutorial for extensions [49]. The first code part shows how the extension sends a message to the content script. As the content script is located at a tab, the id of the tab needs to be specified. In case a content script would send a message to the extension, this id can be left out. The message to be send is then specified along with the callback function handling the response. The second code part shows how the receiver handles the messages - in this case it does not matter weather it is the extension or the content script. The example then simply prints the data and returns a "goodbye" to the sender [49].

```
1 //Sender (background-script)
2 chrome.tabs.query({active: true, currentWindow: true},
  ↪ function(tabs) {
3   chrome.tabs.sendMessage(tabs[0].id, {greeting: "hello"},
    ↪ function(response) {
4     console.log(response.farewell);
5   });
6 });
7
8 //Reciever (content-script)
9 chrome.runtime.onMessage.addListener(
10   function(request, sender, sendResponse) {
11     console.log(sender.tab ?
12                 "from a content script:" + sender.tab.url :
13                 "from the extension");
14     if (request.greeting == "hello")
15       sendResponse({farewell: "goodbye"});
16 });
```

Listing 6.3: Sample messaging between extension and content script [49]

Using messages to pass data to content scripts, they can execute tasks on the DOM like injecting new code or change existing one. Listing 6.4 shows how the background color of the page can be changed to orange once a message with content "changeColor" is recieved in the background

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

script. The API provided by Chrome allows for calling `tabs.executeScript` where the code can be specified as shown in line 6. For bigger changes, or just for the decoupeling of code, this code can also be loaded from a file as shown in line 8 [17].

```
1 chrome.runtime.onMessage.addListener(
2     function(message, callback) {
3       if (message == "changeColor"){
4         chrome.tabs.executeScript({
5           //EITHER
6           code: 'document.body.style.backgroundColor="orange"'
7           //OR
8           file: 'contentScript.js'
9         });
10    }
11 });
```

Listing 6.4: Usage of content scripts as shown in [17]

The functionality described in this subsection only provides a basic overview of a Chrome extension and Chrome's API. For further reading and more information on Chrome extensions, the official developer website from Google is recommended [13].

6.1.2 WebExtension API Polyfill

Besides the Chrome browser extensions discussed in the previous subsection, several other vendors and browsers exist. While many of the commonly used browsers are based on Chromium, an open-source browser project, which is, besides others, also the base for Google Chrome, Microsoft Edge and Opera with a combined market share of about 75 percent [19], and therefore have the same fundamentals when it comes to developing extensions, some vendors implement their browser completely on their own. Even though the basic structure of extensions and the API of the browser is quite similar to the one from Chrome as discussed above, some differences can be found which require adaptions for providing support for multiple browsers. A W3C Community Group was founded several years ago in order "to facilitate discussions between Web Browser vendors", however due to missing participation the community group has become dormant [10].

The WebExtensions API as used by Firefox is "to a large extent compatible with the extension API supported by Chromium-based browsers such as Google Chrome, Microsoft Edge, and Opera" [11]. There are, however, some differences which were pointed out by Mozilla in their documentation [14] regarding the manifest or their API. One key difference is that while in Chrome the API can be found in the `chrome` namespace as described above, the WebExtensions API uses the `browser` namespace. Another difference is that while Chrome uses callbacks for asynchronous calls, the WebExtensions API uses a promise-based approach.

In order to provide a compatible approach for extensions to work both on Firefox and on Chromium-based browsers, Mozilla provides a polyfill framework on GitHub which overcomes this problem [83]. It allows for using the WebExtensions API in Chrome "with minimal or no changes" [83], however does not provide functionality of the API provided by Firefox but not yet implemented in Chrome. Listing 6.5 shows the code how a content script sends a message to a background script using the WebExtensions polyfill framework, similar to the message conversation shown in 6.3. When using the WebExtensions polyfill framework, the shown code can be executed both in Firefox as well as in Chromium-based browsers like Chrome or Microsoft Edge, which means a market share of browser of about 85 percent as of April 2020 [19].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 //Sender (content-script)
2 function handleResponse(message) {
3   console.log(`Message from the background script:
    ↪ ${message.response}`);
4 }
5
6 function handleError(error) {
7   console.log(`Error: ${error}`);
8 }
9
10 function notifyBackgroundPage(e) {
11   var sending = browser.runtime.sendMessage({
12     greeting: "Greeting from the content script"
13   });
14   sending.then(handleResponse, handleError);
15 }
16
17 window.addEventListener("click", notifyBackgroundPage);
18
19 //Reciever (background-script)
20 function handleMessage(request, sender, sendResponse) {
21   console.log("Message from the content script: " +
22     request.greeting);
23   sendResponse({response: "Response from background script"});
24 }
25
26 browser.runtime.onMessage.addListener(handleMessage);
```

Listing 6.5: Sample messaging between content script and extension [84]

As the base system used for POGito, SeleniumIDE, which will be discussed in the upcoming section, already uses the WebExtensions API Polyfill framework, and to provide Firefox support in later stages, this framework was retained for developing the extension.

6.1.3 Selenium

To ease the communication with the browser in content scripts Selenium can be used, which was already discussed in section 2.1.6. Selenium provides an abstraction layer between the browser and the content script such that a developer only needs to call a method without the need of handling browser specifics. The SeleniumIDE, which was used for the base of the prototype, already uses a wide range of these methods and therefore includes all required dependencies to use it in a browser extension. These files are either included in the project as files or loaded into the project during the project setup. One of the most relevant files in the extension is the Selenium BrowserBot, which allows for executing the calls independent of the browser. The BrowserBot allows for easily calling functions like `findElement` to get a web element for the requested locator directly in the code. Furthermore, the SeleniumIDE already includes implementations which abstract the BrowserBot even further, like `doVerifyElementPresent` which either succeeds or throws an error with the corresponding message and is implemented in a file called Selenium-API. While this implementation also calls the BrowserBot, it allows for reduction of the required code for handling repeating calls and their results.

When it comes to the usage of the BrowserBot in the tool, the developer creates a new instance of the Selenium-API with a window which should be used for the execution of the function. Using

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

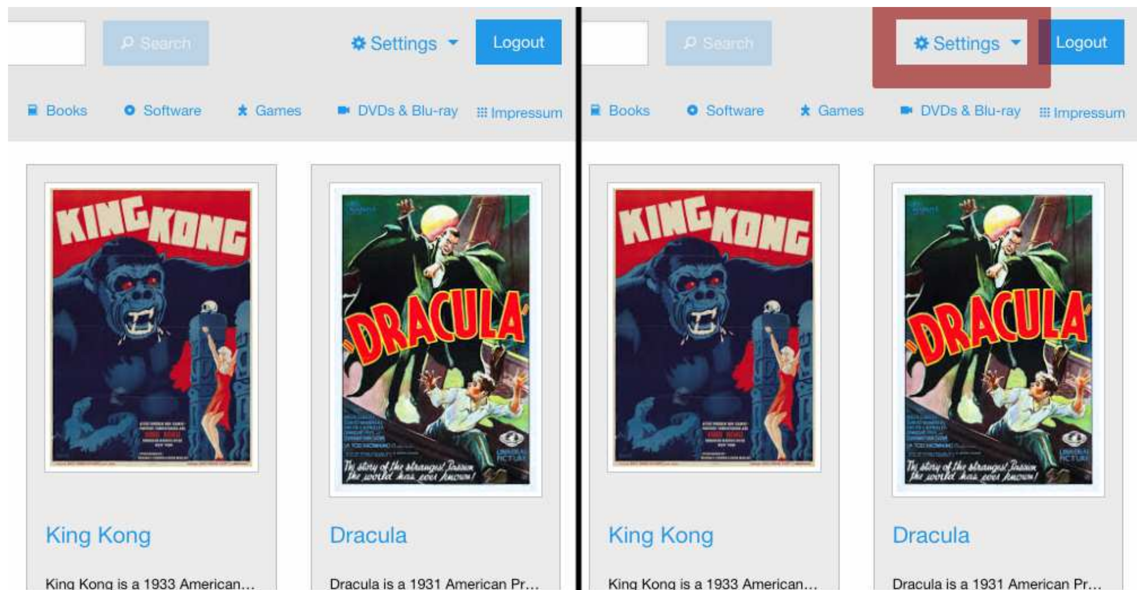


Figure 6.1: Screenshot of a web page in normal state (left) and during the highlighting of an element in the top right corner (right)

this instance, either the BrowserBot can be accessed directly to call the functions or the function provided of the Selenium-API used. Listing 6.6 displays a basic implementation of how the SeleniumIDE shows a single WebElement on the page. The listener for messages to this function is set in line 16. Once a message with content `showElement` is received, the Selenium-API is called on `doShowElement` with the specified target of the message. The result of this call is then the content of a resolved Promise, a concept of asynchronous calls generally used in JavaScript. The function of `doShowElement` is shown partly starting in line 18. This function mainly loads a CSS file, injects it into the current page, and calculates the area of the element which shall be highlighted. After scrolling to the location where the element can be found, the injected border around the element is shown for a few seconds (as defined in the CSS file) and fades away after this timeframe. In the end, this injected element is removed from the page as well. Figure 6.1 shows how a web page is shown to the user when highlighting an element (right) – in this case the menu in the top right corner of the page – in contrast to the normal web page (left).

6.2 POGito

This section describes POGito, the prototype developed for an evaluation of the defined requirements, more deeply. Subsection 6.2.1 provides an overview of the tool with general information on the tool, containing information how it was developed and what was taken into account during development. Furthermore, a screenshot will be provided and discussed, enabling the reader to receive a better picture of the prototype as presented to the experts in the evaluation. Following that, subsection 6.2.2 gives a more detailed and technical insight into the tool by describing how the application is structured internally. Subsection 6.2.3 then presents two selected features and how they are implemented throughout the prototype.

6.2.1 POGito in general

As decided before, POGito was developed as a browser extension written in JavaScript. The main focus was laid on running in the Google Chrome browser. Other Chromium-based browsers and Firefox might also be supported but have not been evaluated. Furthermore, the extension has only

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

been developed and used on Google Chrome on MacOS 10.15 and might not work as expected on other platforms.

```
1  //commands-api.js
2  function startShowElement(message) {
3    if (message.showElement) {
4      try {
5        const result = selenium['doShowElement'](message.targetValue)
6        return Promise.resolve({ result: result })
7      } catch (e) {
8        ...
9      }
10   }
11 }
12
13 if (!window._listener) {
14   browser.runtime.onMessage.addListener(startShowElement)
15 }
16
17 //selenium-api.js
18 Selenium.prototype.doShowElement = function(locator) {
19   const elementForInjectingStyle = document.createElement('link')
20   elementForInjectingStyle.rel = 'stylesheet'
21   elementForInjectingStyle.href = browser.runtime.getURL(
22     '/assets/highlight.css'
23   );
24   (document.head || document.documentElement).appendChild(
25     elementForInjectingStyle
26   )
27   const highlightElement = document.createElement('div')
28   highlightElement.id = 'selenium-highlight'
29   document.body.appendChild(highlightElement)
30   if (locator.x) {
31     highlightElement.style.left = parseInt(locator.x) + 'px'
32     highlightElement.style.top = parseInt(locator.y) + 'px'
33     highlightElement.style.width = parseInt(locator.width) + 'px'
34     highlightElement.style.height = parseInt(locator.height) + 'px'
35   } else {
36   ...
37    }
38   ...
39   scrollIntoViewIfNeeded(highlightElement, { centerIfNeeded: true })
40   highlightElement.className = 'active-selenium-highlight'
41   setTimeout(() => {
42     document.body.removeChild(highlightElement)
43     elementForInjectingStyle.parentNode.removeChild(elementForInjectingStyle)
44   }, 500)
45   return 'element found'
```

Listing 6.6: Sample code of the highlighting of an element as used in POGito

The prototype was build based on a fork of the widely known browser extension called Selenium IDE, which is openly available on GitHub [65]. However, many parts of the extension have been rewritten or adapted to fit the prototype's needs along with newly developed features required for the page object aspect. Selenium IDE provides a Capture & Replay approach for testing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

user interfaces on the web along with a GUI allowing to manipulate these tests or export them as unit tests to a project. Yet, this tool does not provide support for using page objects in the exported files and writes the corresponding paths directly into the tests. At the time of writing, Selenium IDE is ported to be a standalone application, however, the browser extension is still maintained. This browser extension has been chosen as it already provides a wide spectrum of functionality which could be reused for the prototype as well. This includes especially the base setup of browser communication, allowing for reuse of functionality to select or evaluate paths in the DOM. Furthermore, the user interface could be widely reused, allowing saving of time only developing new components as well as providing users with an already familiar UI to facilitate getting started.

The user interface of POGito was developed in React [58], which was already used in the Selenium IDE as well. Further dependencies the tool used contained quality-assuring ones like `eslint` and `stylelint` or build-related ones like `lerna` or `babel`. While some dependencies were removed from the project, as they were not seen to be relevant for the prototype, the ones mentioned along with some others were kept as they provide a good maintainability approach to the prototype as well. In addition to dependencies already included in Selenium IDE, two new ones were added to the project: `handlebars` for an easy and maintainable export of the page objects and `jszip` for exporting the project's files using a zip archive. In addition to these dependencies, the Selenium IDE also loads several Selenium projects in order to provide the functionality in the browser. Even though only a part of the dependencies are used in POGito, they have been fully integrated for a better and easier update process.

Figure 6.2 shows a screenshot of POGito after the tool validated the paths of the page objects. Variables with a valid path are shown in green, while the ones which could not be found on the page are displayed in red. The structure of the user interface is widely as planned in chapter 5, but is extended by some additional elements. The reason for this was that the SeleniumIDE already provided those elements which are seen to be useful and supportive to the user. Therefore, for example, the search bar in the page object list was kept to provide this functionality as well. Another element not planned in the final user interface was the Log-View on the bottom of the screen. This view provides a good overview while validating the variables and provides additional information like its status in textual form, supporting the user for an additional status in a readable form as well, and is therefore kept.

All features discussed in chapter 5 have been implemented in the prototype, which implies that all defined requirements are fulfilled as well. Hence, this prototype can be seen to be sufficient for the evaluation of the requirements.

6.2.2 Code structure

To better understand how the application is structured and to understand the differences between the files in the upcoming subsection, this subsection provides insights into the project structure used by POGito. Listing 6.7 shows an overview of the structure but does not include every file and directory. As one can see, the project groups together the content scripts, and provides packages for the background script, the user interface (ui), the service layer and the models used throughout the project.

The background package contains only one file: the `background.js` file. This file can be seen as the previously discussed background script and includes methods triggered, for example when the extension is started. As this file is the main entry point for the tool, the user interface is also loaded here. Hence, every file in the ui and service package can also be seen as a background script.

The ui package contains user interface related files used in the React application. Its further structure of components and containers is used in the same way as in the SeleniumIDE and follows

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

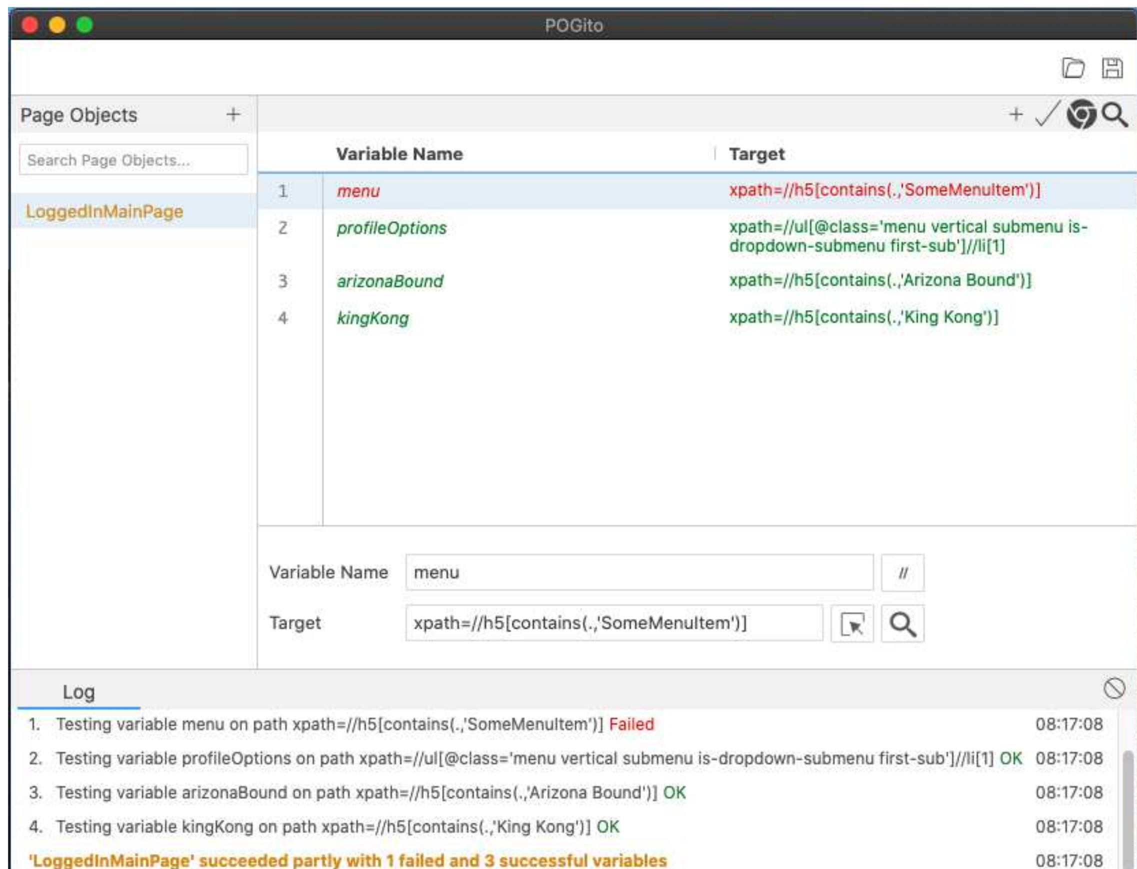


Figure 6.2: Screenshot of POGito after verification of the paths, resulting in three valid and one invalid variables

best practices to break down the user interface into small components. The containers package consists of different general components like the root component as the starting point of the application, the panel component which includes every other component and the navigation component representing the view on the left side of the screen. The components package groups together the smaller components used in the container or in other components. These small parts are, for example, buttons with a specific design, row entries for the page object list and the page object list itself. Each component and container also consists of a style file with specific styling for the corresponding element. General style sheets can be found in the styles package within the ui package. Actions taken by the user in the user interface trigger functions in the service package, which includes, besides others, the file handling for import and export, the find and select functionality or the verification logic of variables. These actions are executed on instances of files which can be found in the model package. This package, for instance, contains the description of a page object in the file `PageObject.js`. Using these files in the package, the whole state of the extension is stored and handled throughout the whole life cycle. Further files in the models package are, for example, the `PlaybackState.js` which stores the temporary and final results of a verification or the `UiState.js` file, which allows receiving user interface related information throughout all components.

While some of the files in the service package only require the context of the application – like importing or exporting – some trigger actions on the web page itself. Content script files can be found in the content packages and are triggered through messaging through background scripts as already mentioned. The `commands-api.js` file is the main entry point and receives every message send to the content script from the tool and handles the corresponding action before answering

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

the result back through messages to the tool. Files in this package also contain functionality for displaying the element on the page or for selecting an element and receiving its targets.

```
1  |--- background                           |--- ui
2  |   |- background.js                      |   |--- components
3  |--- content                              |   |  |--- ActionButtons
4  |   |- commands-api.js                    |   |  |--- PageObjectList
5  |   |- locatorBuilders.js                 |   |  |  |- index.js
6  |   |- targetSelector.js                  |   |  |  |- style.css
7  |--- model                                |   |  |--- PageObjectRow
8  |   | --- Command                         |   |--- containers
9  |   |   |- index.js                       |   |  |--- Editor
10 |   |- PageObject.js                      |   |  |  |- index.js
11 |   |- PlaybackState.js                   |   |  |  |- style.css
12 |   |- ProjectStore.js                    |   |  |--- Panel
13 |   |- UiState.js                         |   |  |--- Root
14 |--- service                              |   |--- styles
15 |   |--- import-Export
16 |   |  |--- languages
17 |   |  |    |--- java
18 |   |  |    |    |- handlebarsHelper.js
19 |   |  |    |    |- javaExporter.js
20 |   |  |    |    |- javaImporter.js
21 |   |  |- filesystem.js
22 |   |--- IO
23 |   |  |- find-select.js
24 |   |  |- recorder.js
```

Listing 6.7: Package structure of POGito (not complete)

6.2.3 Selected implementations

This subsection will describe two selected features and how they are implemented in a more detailed way. The first feature is about the export process and describes how POGito creates Java page object files out of its internal structure. The second feature shows how a single element on the DOM tree is highlighted if requested by the user.

Java Export

One major feature when supporting the maintenance of page objects was the import from and the export to Java files. As POGito internally holds an object representation, as already discussed in 6.2.3, this representation needs to be converted to a file in Java format. While the generation of these files could have been done manually, a more maintainable and language-exchangeable approach was chosen by using Handlebars. Handlebars is a library for JavaScript but was also ported to other platforms. It compiles semantic templates into JavaScript functions such that only these functions need to be called for creating the file content [31]. This file content then only needs to be written to a file to receive a valid Java page object. Handlebars is based on mustache, an open-source framework with support for more programming languages than JavaScript [50]. In the following, the process of exporting a file will be described in more detail:

When selecting a single file export in POGito – or when only one page object is maintained in the tool – the method `exportPageObject` as shown in listing 6.8 is called. By allowing for providing the parameter `fileType`, these methods can easily be extended for further programming

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

languages, while Java is currently set to be the default. Furthermore, this method requires the selected page object as parameter, which allows a loose coupling between the services. This method then calls the method `exportJavaAsSingleFile` function, which is also shown in listing 6.8. This function is responsible for handling of the creation of the necessary files and triggering the download function. It also handles the creation of the valid Java class name which should by definition start with a capital letter and not contain spaces.

```
1 export function exportPageObject(pageObject, fileType = 'java') {
2   if (fileType === 'java') {
3     exportJavaAsSingleFile(pageObject)
4   }
5 }
6
7 export function exportJavaAsSingleFile(pageObject) {
8   let exportPageObjectName = pageObject.name
9   let exportPageObject = createPageObject(pageObject)
10
11  downloadUniqueFile(
12    createValidJavaName(exportPageObjectName, false) + '.java',
13    exportPageObject,
14    'text/x-java-source,java'
15  )
16 }
17
18 function createPageObject(pageObject) {
19   let templateScript = Handlebars.compile(getJavaObjectTemplate())
20
21   Handlebars.registerHelper('escape', function(variable) {
22     return variable.replace(/(['"])/g, '\\$1')
23   })
24
25   let context = {
26     packageName: pageObject.packageName,
27     classExtensions: pageObject.classExtensions,
28     importStatements: pageObject.importStatements,
29     className: createValidJavaName(pageObject.name),
30     variables: createVariables(pageObject.commands),
31   }
32
33   return templateScript(context)
34 }
```

Listing 6.8: Sample code of the export function as used in POGito

The main creation of the file content happens in the function `createPageObject` which takes a page object for generation. Listing 6.8 shows the function which compiles the Handlebars template shown in listing 6.9 and triggers it with a context containing all relevant data for the page object, like the variables – with name and path – along with import statements and other related information. It should be noted that the content is provided as a JSON, which is than parsed by Handlebars. By using Handlebars, different paths can be displayed or repeated without much need of logic, depending on the context provided and the statement in the template itself. Furthermore, this allows for an easy exchange of the template to other file styles, potentially even by the user. In the current implementation, POGito only supports the described annotation approach.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Finally, the `downloadUniqueFile` function is called, a function already provided by SeleniumIDE, which handles the creation and download of the file with the given name, the given content and the mime-type of a java file. The user is presented with a download dialog as commonly known in a web browser and can then download the file as from every other web page.

```
1  '{{#if packageName}}' +
2  '{{packageName}}\n\n' +
3  '{{/if}}' +
4  '{{#if importStatements}}' +
5  '{{#each importStatements}}' +
6  '{{this}}\n' +
7  '{{/each}}' +
8  '{{else}}' +
9  'import org.openqa.selenium.WebDriver;\n' +
10 'import org.openqa.selenium.WebElement;\n' +
11 'import org.openqa.selenium.support.FindBy;\n' +
12 '{{/if}}' +
13 '\n\n' +
14 'public class {{className}} ' +
15 '{{#if classExtensions}}' +
16 '{{classExtensions}}' +
17 '{{else}}' +
18 'extends PageObject' +
19 '{{/if}}' +
20 ' {\n' +
21 '{{#each variables}}' +
22 '{{#with this}}' +
23 '\n' +
24 '   {{comment}}@FindBy({{prefix}} = "{{{target}}}")\n' +
25 '   {{comment}}private WebElement {{variableName}};\n' +
26 '{{/with}}' +
27 '{{/each}}' +
28 '\n' +
29 '   public {{className}}(WebDriver driver) {\n' +
30 '       super(driver);\n' +
31 '   }\n' +
32 '\n' +
33 '   //Insert your methods here\n' +
34 '}'
```

Listing 6.9: Handlebars template used for Java export

The result of the export, which was triggered in a state of POGito as shown in figure 6.3, can be seen in listing 6.10. As one can see, all variables are exported to a Java page object based on the defined annotation structure of the handlebars template in listing 6.9. The variable *arizonaBound* was exported as commented out to display this functionality as well. The package path and the imports have been reused from the imported page object and only the methods have been removed to provide a valid page object.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

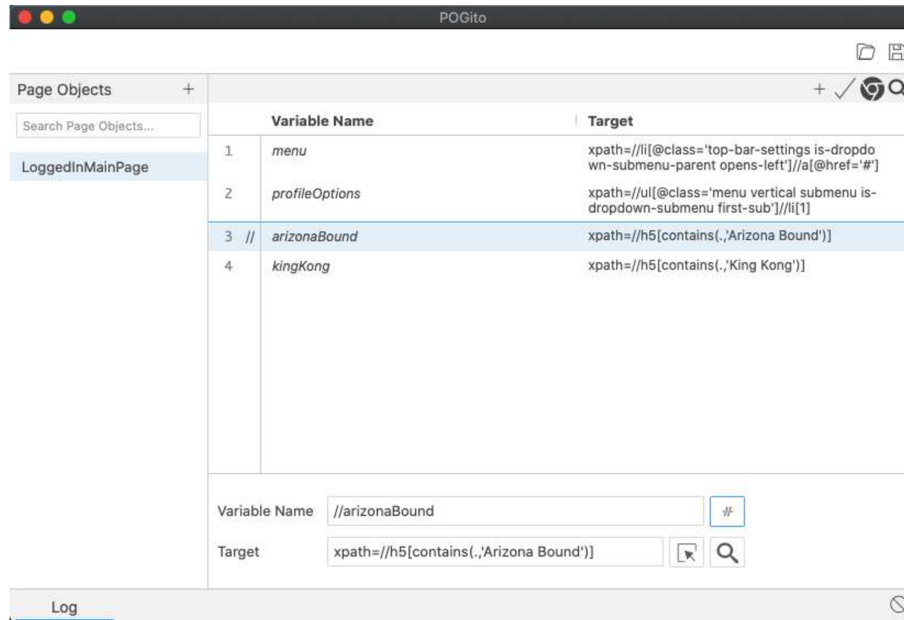


Figure 6.3: Page object in POGito before the export

```
1  package at.ac.tuwien.inso.swtesten.lab.pages;
2
3  import at.ac.tuwien.inso.swtesten.util.PageObject;
4  import org.openqa.selenium.WebDriver;
5  import org.openqa.selenium.WebElement;
6  import org.openqa.selenium.support.FindBy;
7
8  public class LoggedInMainPage extends PageObject {
9
10     @FindBy(xpath = "//li[@class='top-bar-settings
       ↪ is-dropdown-submenu-parent opens-left']//a[@href='#']")
11     private WebElement menu;
12
13     @FindBy(xpath = "//ul[@class='menu vertical submenu
       ↪ is-dropdown-submenu first-sub']//li[1]")
14     private WebElement profileOptions;
15
16     //@FindBy(xpath = "//h5[contains(.,'Arizona Bound')]")
17     //private WebElement arizonaBound;
18
19     @FindBy(xpath = "//h5[contains(.,'King Kong')]")
20     private WebElement kingKong;
21
22     public LoggedInMainPage(WebDriver driver) {
23         super(driver);
24     }
25
26     //Insert your methods here
27 }
```

Listing 6.10: Exported page object of POGito

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

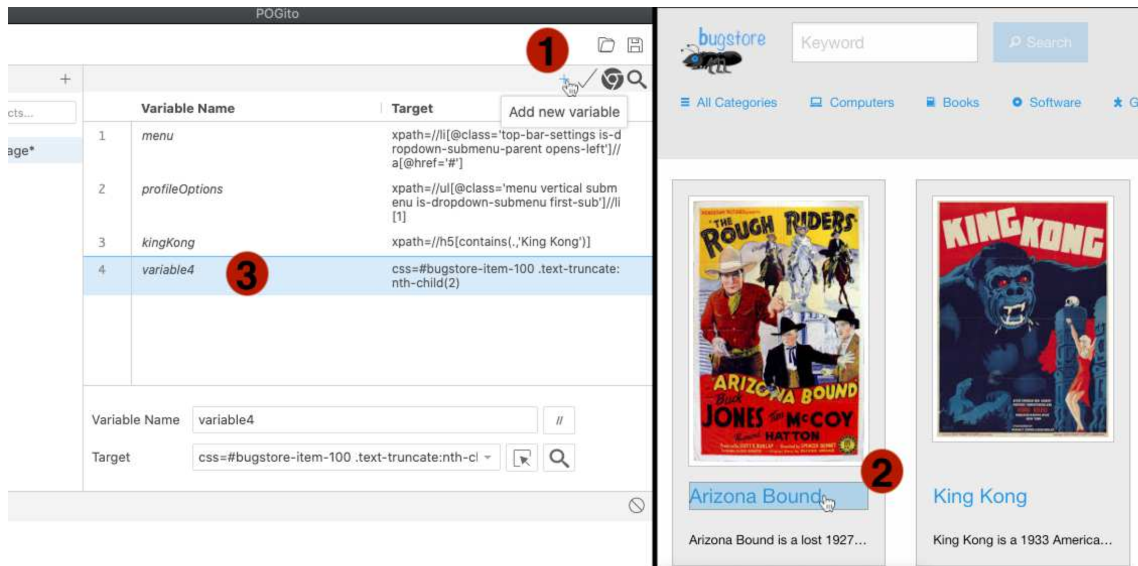


Figure 6.4: Screenshot of POGito (left) and the browser (right) during adding a new variable

Adding a new element to the page object

One defined requirement mentions that it should be easy to add new variables to page objects. As SeleniumIDE is a capture and replay tool, the capture function already provides the basic functionality for recording the user's clicks on a web page. This functionality was adapted to only record click events on elements on the page instead of every interaction. Furthermore, the process was adapted to only prompt the user for exactly one click, add this element as a variable to the page and stop the recording process. In the SeleniumIDE, the process records every action until the user stops the recording process manually. The functionality of highlighting the element which would be selected was kept in order to provide better feedback to the user. Figure 6.4 shows how the user sees the web page while adding a new variable and furthermore the steps taken to add the element to the page object. First the user clicks on the "Add new variable" button in POGito as shown at step 1, which focuses the browser window and highlights elements the user hovers over. The right image of figure 6.4 shows that the variable "Arizona Bound" is highlighted at step 2. Once the user clicks on that element, POGito is focused again and the variable is added using a auto-generated name, which is shown at step 3.

When the button for adding a new variable is clicked in the extension, first the recorder defined in `recorder.js` is started to record on the current window. The recorder uses different listeners for events in the browser, like adding or removing tabs to handle changes correctly. These listeners use the API of the WebExtension polyfill to be triggered by the browser. The recorder has been adapted to stop recording once the first click was recorded. After the recorder was started, the browser window is focused and a message send to the content script in `find-select.js` to start the selection process. Relevant parts of the `select` function are shown in listing 6.11 along with the `doCommands` function of the content script (`commands-api.js`) which received these messages and handles the further selection process.

When the content script is called, a new `TargetSelector` instance is instantiated. This instance takes a callback function as well as a cleanup callback function and then injects a script to the page which handles the recording. The main part of this script consists of two listeners which are added to the page and triggered through mouse moves and clicks on the page. These listeners then trigger the `handleEvent` function as shown in 6.12. While the mousemove listener requests the element of the given coordinates from the DOM, as shown in the `highlight` function be-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

low, and highlights it by adding a CSS background to it, the click listener is triggered once the highlighted element is clicked and calls the defined callback function with this element. The last action taken by the content script is building the targets using the `LocaltorBuilders`. The callback function calls the `buildAll()` function in line 25 in listing 6.11 with the provided element. The `buildAll()` is included in the SeleniumIDE and builds all the possible targets for the given element. This array of targets is then reported back to the background script.

```
1  //In background script
2  export async function select(createCommand = false) {
3     ...
4     await browser.windows.update(tab.windowId, {
5        focused: true,
6     })
7     await browser.tabs.sendMessage(tab.id, {
8        selectMode: true,
9        selecting: true,
10       element: true,
11       selectNext: false,
12    })
13    ...
14 }
15
16 //In content script
17 function doCommands(request, _sender, sendResponse) {
18 ...
19 if (request.selectMode) {
20    sendResponse(true)
21    if (request.selecting && request.element) {
22      targetSelector = new TargetSelector(
23        function(element, win) {
24          if (element && win) {
25            const target = locatorBuilders.buildAll(element)
26            ...
27               browser.runtime.sendMessage({
28                 selectTarget: true,
29                 target: target,
30                 selectNext: request.selectNext,
31               })
32          }
33          targetSelector = null
34        },
35         ...
36      )
37    }
38 ...
39 }
40 browser.runtime.onMessage.addListener(doCommands)
```

Listing 6.11: Sample code of the select process for a new varibale

While the previous code and process is taken directly from SeleniumIDE – except for `recorder.js` – the process has been changed from the point when the result of the content script is received in the background script (`find-select.js`). Depending on whether the select was called for adding a new variable or as support for changing the targets from an existing one, the targets are added

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

to a new instance of a variable or an existing one. In case the variable is newly added, a default name is set to better differentiate the variables in the user interface. As the target is provided as an array, the first target is chosen to be the target for this variable, however, the array is saved as well to provide the user with additional targets if required. Once the variable is saved to the project, a message is sent to the content script for cleanup and the process of recording is stopped.

```
1
2 handleEvent(evt) {
3     switch (evt.type) {
4       case 'mousemove':
5         this.highlight(evt.target.ownerDocument, evt.clientX,
          ↪ evt.clientY)
6         break
7       case 'click':
8         if (evt.button == 0 && this.e && this.callback) {
9           this.callback(this.e, this.win)
10        }
11        ...
12        break
13    }
14  }
15
16  highlight(doc, x, y) {
17    if (doc) {
18      const e = doc.elementFromPoint(x, y)
19      if (e && e != this.e) {
20        this.highlightElement(e)
21      }
22    }
23  }
```

Listing 6.12: Excerp of the TargetSelector (in targetSelector.js)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

7 Evaluation

This chapter has its focus on the evaluation of the requirements to an approach supporting the page object maintenance defined in chapter 4. This evaluation was conducted as an expert evaluation based on the developed prototype called POGito described before. For this evaluation, testers with several years of experience in the field of software testing have been chosen as experts to answer a questionnaire. Section 7.1 will describe the conducted evaluation process containing a questionnaire, participant description and general factors of the environment. Following on that section 7.2 will describe the threats to validity, which should be taken into account when reading the evaluation results which are presented and discussed in section 7.3.

7.1 Evaluation process

In the following section, the evaluation process is explained. This includes all relevant information on the expected goal as well as the steps taken and the environment used to achieve this goal. For describing the steps used, this section also describes the scenarios demonstrated in the interview and their relation to the requirements defined before as well as the questionnaire used in the interview based on which the results in the following section are discussed.

7.1.1 Goal

The goal of the conducted interviews lies in the evaluation of the requirements defined in 4.2 and the classification of them to either be useful or not in the eyes of the asked experts. Furthermore, additional requirements shall be gathered in order to enhance the maintainability approach to page objects further. By doing so, research question 3 as defined in section 1.2 is answered.

It should be noted that the goal was to evaluate the requirements without the direct effect of the usability of the prototype. Therefore, each interaction with the prototype was done by the interviewer to receive answers based on the functionality. If the user could have done the interaction on its own, requirements could have been rated lower due to potential bad implementation of the prototype or higher due to a unintended behavior.

7.1.2 Participant characterization

The participants of this evaluation have been chosen and were provided by the research group for Industrial Systems (INSO) and consisted of five experts working on different projects. All of the participants have a multi-year experience in software engineering in general as well as a multi-year experience in the field of software testing and especially using the page object pattern in multiple projects.

7.1.3 Evaluation environment

Due to the exceptional situation during writing this thesis – having a lockdown of all public relations due to COVID-19 –, the evaluation was conducted via video calls. All interactions with the prototype were done by the interviewer on a MacBook Pro using Chrome for the prototype and IntelliJ for simulating the production environment. The screen of the interviewer was broadcasted to the interviewee using the screen sharing option during the whole interview.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

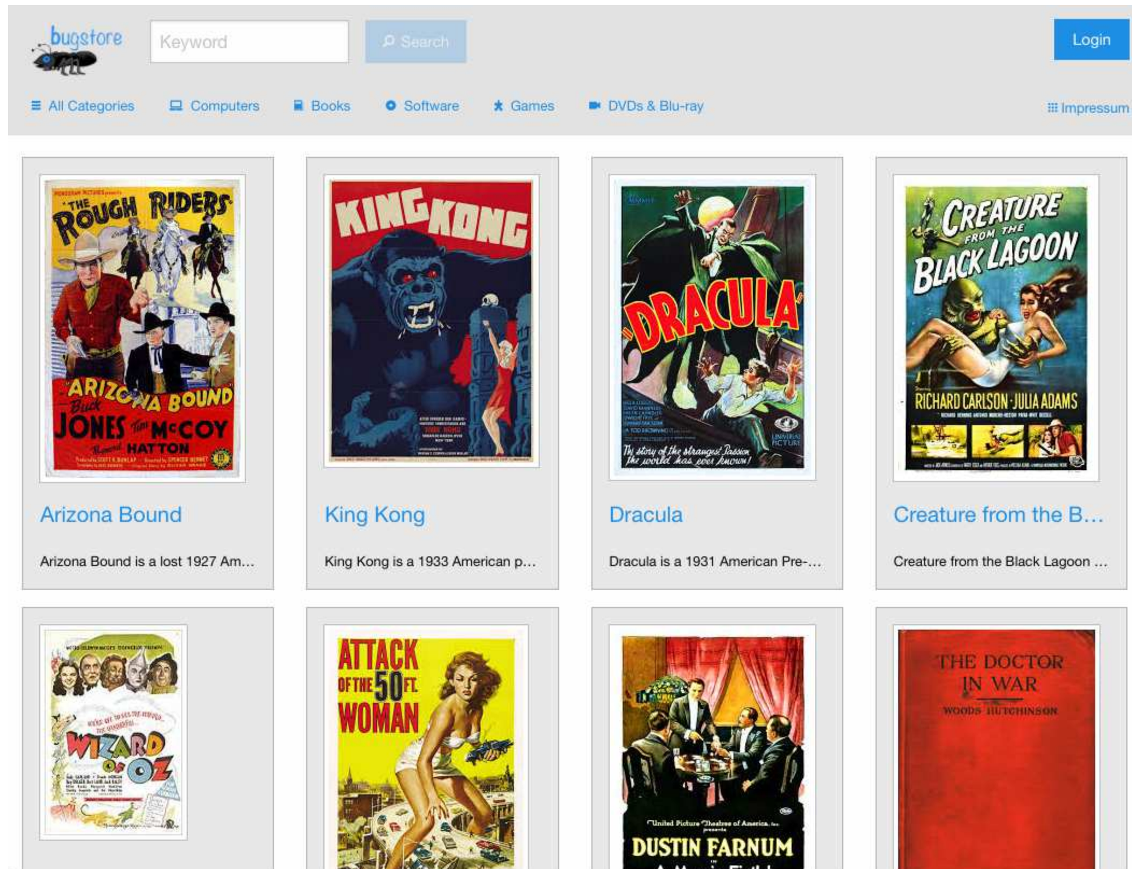


Figure 7.1: Screenshot of the bugstore

The web application used for evaluation – a small online store called bugstore – can be seen in a screenshot in figure 7.1. The application was chosen for the evaluation as it is representative for web applications used in production environments concerning features, behavior and how it can be tested on a system level. Furthermore, as the bugstore is a custom build web application for university purposes and controlled by INSO, an expected state of the application could be easily established for demonstration purposes.

Based on the web application, a small sample test case was implemented which ought to reflect the real-life scenarios tester face in their daily routines. The steps of the test case were simple, including just navigations through the pages with entering some information, but sufficient to demonstrate each scenario of the prototype. The test was intentionally created having some downsides in the implementation, in order to demonstrate the problem solving using POGito during the scenarios. As already mentioned, the test case was executed in IntelliJ during the interview.

7.1.4 Methodology

In order to receive comparable feedback by the experts, every interview was done equally and as described in this subsection. However, it should be noted that the experts were allowed to disrupt the demonstration of the scenarios at any time and comment or ask a question, resulting in slightly different results during that phase. There was no fixed duration of the interview given as discussions throughout the interview were welcomed, but would have interfered with the time schedule. On average, an interview lasted about one hour. As the structure of the interviews were all the same they could be divided into the following main parts:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Joining and general introduction
- Pre-demonstration questionnaire
- Goals and topic introduction
- Scenario demonstration
- Post-demonstration questionnaire

In the following, each part will be described more deeply.

Joining and general introduction

First, a video meeting was set up and provided by INSO where both the interviewer and interviewee joined. As already mentioned, the screen was shared by the interviewer throughout the session, providing the interviewee with the following applications during the interview:

- POGito, the prototype as Chrome extension, developed in this thesis and used for evaluation.
- IntelliJ, an IDE (Integrated Development Environment) used for displaying the source code and executing the test.
- Chrome, a web browser serving several purposes:
  - As a starting point for POGito
  - For the introduction to the web application bugstore
  - To display the questionnaire to the interviewee during the questioning phases

The application shown on the screen depended on the current situation and changed throughout the interview frequently.

Additionally, some short clarifications of formalities like the language of the demonstration and the consent to the recording were discussed. This also formed the base of conversation for both parties to get to know each other.

Pre-demonstration questionnaire

The interview began by asking the pre-demonstration questionnaire. The questions asked were shown to the expert through screen sharing and were asked by the interviewer as well. The goal of the questions was to evaluate the position and knowledge of the expert as well as their current opinion on page objects, including downsides they see or encounter in their daily work routines. Answers were either expected as a number between 1 and 4 (quantitative), where 1 was seen as the most negative option and 4 as the most positive one throughout all questions, or as an open answer without limitations (qualitative). During the pre-demonstration phase, the following questions were asked by the interviewer:

1. Please give an overview of your daily work. What is your job position? Which test-types are most used by you? How are you dealing with page objects in your tests? Which development process (agile, waterfall. . .) is most used when you are testing?
2. On a scale from 1 to 4, how would you rate your experience with page objects?
3. Did you encounter any problems using page objects yet? If yes, which?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4. How are you currently maintaining your page objects?
5. Do you currently use any tools for maintaining page objects? If yes, which?
6. On an average project you are working on, how often do you have to ...
    6.1. ... maintain (update) a small subset of your page objects?
    6.2. ... maintain (update) almost every page object of the project?
    6.3. ... replace page objects completely?

The whole questionnaire as used by the interviewer and shown to the experts on the screen can be found in the appendix A.1.

Goals and topic introduction

After all questions of the pre-questionnaire were asked, the expert was provided with an introduction to the topic which included a general overview on the goal of the thesis as well as a description of POGito and the problems which it should solve.

Furthermore, the expert was provided with an overview of the current state of the test case along with a short introduction in the bugstore application as system under test. This introduction was done interactively and the expert was allowed to ask questions on both objects to make themselves familiar with the application and the test. The interviewer acted as instructed by the interviewee in this stage by showing the requested elements on his screen.

As the last part of this stage, the general situation of the demo was described. This included the background story from the interviewer's viewpoint as follows:

I am a developer new to the team. I was given a test case that navigates to the address site in bugstore and changes it there. The test case is implemented using JUnit and uses page objects for modeling each page in the application. However, this test case is currently not working due to problems in the page object implementation.

Before the demonstration started, a PDF outlining the scenarios was provided to the expert, which included the scenarios' pre- and post-condition along with every step executed during the demonstration. This scenario outline could be used by the expert to reflect each step taken, ask questions if a step was not fully understood or to later refer to certain steps in the questioning phase afterwards. The PDF file as sent to the experts can also be found in the appendix A.2.

Scenario demonstration

Following the introduction, the demonstration of the prototype started. The walkthrough was based on five scenarios which covered the whole functionality POGito provides and was self-contained.

As already described, each scenario was based on a test case which evaluated a demo web application. In each scenario, a problem of the test case was supposed to be fixed such that after all scenarios the test succeeded. A further description of each scenario, its pre- and post-condition, its steps taken, as well as its relation to the requirements can be found in the following subsection.

During this phase, the main goal was for the interviewer to solve the problems of the demonstration project using the prototype, presenting the general use of it along with every step taken to achieve the expected result. The interviewee was allowed to ask questions and for the interviewer to repeat one or multiple steps to gain a better understanding of the usage. In case the question would be answered in a later scenario, the interviewee was informed that this question will be covered in a later scenario and asked to postpone the discussion.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Post-demonstration questionnaire

After all scenarios were presented to the expert, questions about the prototype were asked. They were again designed to include both quantitative and qualitative questions in order to receive comparable values but also the thoughts and expectations of the experts along with general feedback.

The following enumeration lists all questions asked. Again, answers ranging from 1 to 4 reflect 1 as the lowest and lest preferred selection and 4 the highest and most preferred selection. The detailed questionnaire as presented and used for the evaluation can be found in the appendix A.1. In case a question focuses on the evaluation of a requirement defined before in section 4.2, the requirement is added in braces before the question is asked.

7. Based on the demonstration, do you think the provided prototype can enhance your daily work using page objects? If no, why not?
8. (Only asked if answered yes on question 3) Based on the demonstration, do you think the provided prototype can solve the problems you mentioned? If no, why not?
9. Based on the demonstration, do you expect that POGito can handle all your page objects of a project and their variables in a convenient and clear way? If no, why not?
10. On a scale from 1 to 4, how intuitive did you find POGito?
11. On a scale from 1 to 4, how much time of your daily routine when working with page objects could be saved when working with POGito instead of your current approaches.
12. On a scale from 1 to 4, how much would you like to use POGito in your daily work with page objects?
13. On a scale from 1 to 4, how likely would you recommend POGito to your colleagues?
14. Please rate the usefulness of the given feature from 1 to 4:
    - 14.1. (R1, R2) Import/Export of page objects
    - 14.2. (R5, R6) Adding new variables through a GUI based approach
    - 14.3. (R4, R5, R6) Changing or removing variables through a GUI based approach
    - 14.4. (R8) Displaying a single variable on the page, or giving information about its existence
    - 14.5. (R8) Displaying all found variables on the page
    - 14.6. (R7) Verifying all variables of a page object for their existence
    - 14.7. (R3) Support for maintaining multiple page objects at the same time
15. Which feature of POGito did you find the most appealing?
16. Are there any shortcomings in the implementation which you expect to be included in a maintenance tool? If yes, which?
17. Is there anything you would like to add?

Especially the last question was supposed to start a discussion about further required features and general thoughts the interviewee has. Due to the openly asked questions and the way the interview was handled, a lot of additional information could be gathered to understand the needs of experts better and discover a way the tool should proceed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

7.1.5 Scenarios

In the following, each scenario will be described deeply. For better understandability, the scenario will also outline which requirement (as described in chapter 4) and which feature (as described in chapter 5) are covered by the given scenario. As already mentioned, the PDF containing all relevant information as provided to the experts can be found in the appendix A.2. This description, however, will extend the information which can be found there.

Scenario 1

Short description:	I want to add a missing element to the existing page object
Requirements covered:	R1, R2, R4, R5, R6, R7 (partly), R8
Current situation (Pre condition):	- The test stops on the start page as an element is missing
Expected outcome (Post condition):	- I have added the required variable to the page object - The test runs and navigates to the login page
Steps definition:	1) I open POGito 2) On the startup page, I select “Using an existing page object” and select the malfunctional page object in the file chooser 3) I open a browser session in POGito and navigate to the page 4) I click on ‘Show all variables’ on the page 5) I find out that the login button is not covered by the page object 6) I add the menu button to the page object and name it loginButton 7) I export the page objects and fix their methods in Git 8) I run the test to make sure the test navigates to the login page

In this scenario, the expert is introduced to POGito for the first time. The focus of this scenario lies in the update process of a page object as one variable is missing on it. With the help of the inspection feature of POGito providing an overview of all variables on the page, the missing element could be found easily and added through the "Add element" feature of the prototype. In this scenario, the expert also experiences the usage of the functions to import and export a single file.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Scenario 2

Short description:	I want to create a page object for the login page with variables for the username-field, password-field and for the login button
Requirements covered:	R2, R3, R5, R6, R7 (partly), R8
Current situation (Pre condition):	- The test stops on the login page - I do not have a login page modeled as page object
Expected outcome (Post condition):	- I do have a login page modeled containing the required fields - The test proceeds and navigates to the start page in the logged-in state
Steps definition:	1) I create a new page object and name it ‘LoginPage’ 2) I navigate to the login page in the browser 3) I add a variable for username via select of the element and rename it username 4) I add a variable for password via select of the element and rename it password 5) I add a variable for login-button manually by entering the information and name it loginButton 6) I export both files to the test-suite and fix their methods in Git 7) I run the tests and check that it is not failing on the login page anymore

The focus of the second scenario lies in the creation of page objects through POGito. While the process of adding new variables is similar to the previous scenario, one variable is also added manually to demonstrate the option as well. For the manually added variable, the feature for showing a single variable on the page was also demonstrated to verify if the entered target is correct. Furthermore, in contrast to scenario 1, the export was now presented as a zip file containing all page objects, as here multiple files are required.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Scenario 3

Short description:	I want to check if there are any invalid paths on the page and adapt them
Requirements covered:	R1, R2, R3, R4, R5, R6, R7
Current situation (Pre condition):	- The test fails on the start page in the logged-in state - I do not know, if and which of the given paths on the page object are invalid
Expected outcome (Post condition):	- The invalid path of the page object is corrected - The test runs and succeeds
Steps definition:	1) In POGito I open the page object file for the start page in the logged-in stage 2) I log myself in to the application in the browser and navigate to the start page 3) I click on verify variables 4) I can see one entry marked as invalid 5) I click on the invalid entry, us the button “Select target” and choose the correct target on the page 6) I export the given page object back to the original place 7) I adapt the page objects methods via Git 8) I run the test and check that it succeeds

The focus of this scenario lies in the evaluation of invalid variables. Therefore, the feature of POGito providing an overview of the states the variables are in (found/not found) is used to identify invalid variables. After finding the invalid variable, it was also shown how the variable can be updated through the tool either by selecting it or by manually entering the new value. Lastly, the "Copy to clipboard" feature was presented, which the page object could be easily copied back to the IDE with.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Scenario 4

Short description:	I want to adapt a path on the start page
Requirements covered:	R1, R2, R4, R6, R7 (partly)
Current situation (Pre condition):	- The test runs successfully - I am not satisfied with the selected path of the menu item
Expected outcome (Post condition):	- I have adapted the path of the menu item - The test still runs
Steps definition:	1) I check that I am still on the main page in logged-in state in POGito as well as in the browser 2a) I change the path by using the dropdown menu OR 2b) I change the path manually 3) I check if the path is valid by finding the element on the page 4) I export the page objects and fix their methods in Git 5) I run the test to make sure the test still succeeds

As the tests are now succeeding, this scenario deals with modifying the page object to the users' needs. During this scenario, the expert was presented with the options to change the target of an element using either the text field to manually enter the new target or to use predefined values from POGito to select from through the dropdown menu.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Scenario 5

Short description:	I want to check if all variables are required on the start page
Requirements covered:	R1, R2, R4, R6, R7 (partly), R8
Current situation (Pre condition):	- The test runs - Sonar gives me a message that two variables are unused - I am not sure if I still need these variables
Expected outcome (Post condition):	- I have found out that I do not need both variables in my test - I have removed the first variable from the page object - I have commented the second variable on the page object - The test still succeeds
Steps definition:	1) I check that I am still on the main page in a logged-in state in POGito as well as in the browser 2) I click on 'Show all variables' on the page 3) I find out that two variables are not needed by my tests 4) I remove one variable via POGito, as will not use it in the near future 5) I comment one variable out via POGito, as I expect to use it in the near future 6) I export the page objects and fix their methods in Git 7) I run the test to make sure the test still succeeds

The last scenario also focused on the modification of variables on a given page object. In this case, the variables were either deleted or commented out to not include them in the evaluation of POGito or in the exported page object in case of the deleted variable.

Combining all scenarios, each requirement described in section 4.2 was covered at least once in order to evaluate them in the questionnaire. Figure 7.2 provides an overview of which scenario covered which requirement. A green cell means that the requirement was covered in this scenario, a red cell shows that the requirement was not covered in the scenario, a yellow cell means that a requirement is partly covered in this scenario.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

	S1	S2	S3	S4	S5
R1: Reuse of existing page objects					
R2: Maintaining only relevant page objects					
R3: Overview of multiple page objects					
R4: Maintaining only relevant variables					
R5: DOM identification					
R6: Adding and modifying variables of page objects					
R7: Overview of variables and their paths validity					
R8: Display page object variables on page					

Figure 7.2: Scenarios and the defined requirements they address

7.2 Threats to validity

This section deals with the treads to validity of this evaluation which should be taken into account when reading about the results.

Number of participants

For the given evaluation, only five experts were asked to provide feedback to the prototype due to time limitations of a master thesis. While this allows for answering the questions, evaluating the chosen requirements and gathering further requirements, it does not allow for providing a wider overview of different needs software testers have to an application as described.

No expert interaction

As already described before, the goal of the evaluation was to verify the requirements without the interaction of a user to prevent usability problems having an effect on the results. This results in a missing personal opinion of the experts relating to the usage of the tool, which might have an impact on the result or extend it further.

Predefined web application

The provided web application served well for the demonstration purposes and was chosen as it included all relevant elements of a general web application. However, due to the wide variety of web application frameworks and implementation styles, the prototype might or might not work on these applications. The results of the evaluation can only be seen in the context of this specific application and need further evaluation on different other pages.

Predefined test suite and scenarios

Even though the test suite has been designed to be representative for common test automation cases, the scenarios were held general and chosen on a common base. Experts might have test suites or page objects which do not work as shown in the evaluation. While this is should not have an impact on the results regarding the requirements, testers might have further, more project-specific, requirements to add.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

7.3 Evaluation results

Based on the interviews with the experts as described in subsection 7.1.4, this section will now give an overview of the results received. Before discussing the feedback on the prototype, the results of the questionnaire asked before the demonstration will be discussed in order to provide an overview on the experts.

7.3.1 Results of the pre-demonstration questionnaire

The goal of the first question in the questionnaire was to gain a general overview on the interviewee and to decide whether or not the person can be called an expert in the field. The experts provided insights into their daily work life, the projects they are currently working on and projects which have been already finished. While two of the five experts are full-time software testers, the other three are also software developers but with a strong relation to testing. This relation is shown as even though they are developers in their project, they also play major roles in the project's testing part. Furthermore, it should be noted that interviewees mentioned a very broad usage of page objects. The systems tested by the experts ranged from Java and Selenium tests over Angular web applications to a strong mobile background using Appium for tests. Based on that, not every expert could use the prototype in their current project, however, the requirements could be evaluated as the general concept could easily be adapted. On average, the experts rated their experience with page objects with 3,4. Based on that rating and the fact that the interviewees have multiple years of experience in the topic of software testing with page objects, all of them could be seen as experts in the field and therefore relevant for the evaluation.

When talking about the way page objects are currently maintained and tools used for the maintenance, the most common answer was the manual interaction by the tester or developer. Most adaptions are made using the browsers integrated developer tools by inspecting the DOM tree, picking the correct target and modifying the page object manually. One expert, whose developers are also maintaining the page objects due to the size of the team, also mentioned that as the developer writes the code on the web page, the correct target can be easily chosen without the need of much inspection. However, some tools are also used, especially for enhancing the targets to be more robust. One mentioned tool is the ChroPath browser extension which automatically generated paths for the user. Another expert mentioned an adapted reporting tool which generates reports in a way that invalid paths can easily be detected by the testers in case a test fails.

The experts were also asked to define problems they currently face when using the page object pattern. The answers can be split into two main categories: social and knowledge problems on the one hand and technical problems on the other. When talking about the social and knowledge aspect, one interviewee mentioned a missing convention for designing page objects. Therefore the expert often encounters different page object styles, making it hard to understand the structure and maintain it further. The same expert also mentioned that page objects are often implemented more than once due to missing communication which also leads to problems when maintaining them, as all classes of the page need to be found in the project. Too much logic in the page object is also a problem encountered in the daily routine. On the technical side, the problems mentioned were similar to the ones already discussed in the problem description and experts often named the dynamism of web pages – sometimes in a relation to specific web frameworks – as the main problem when designing them. A not yet discussed problem experts face are complex web pages which also need to be designed in the page objects. These design process includes either inheritance of the objects or defining sub-components of the web page as single page object such that the test uses multiple ones. While this allows for easily covering complex web pages, this also makes the maintenance of page objects harder.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The last question asked before the demonstration had its focus on how often the page objects need to be maintained and how often they are completely rewritten. For simplicity, the interviewee talked about their current project and its characteristics, which resulted in two different groups of answers as some projects were in the initial development phase and some already in the maintenance or enhancement phase. For those in the initial development phase, experts mentioned that the maintenance of their page object consists of adding functionality to them such that a small subset is maintained very often, but almost no recreation of them is needed. For those in later phases of development, again two groups could be identified. One group mainly maintained their page objects by updating their files and targets once they change and only recreated them after major changes while the other group of experts mainly recreated their page objects from scratch once some of the paths in it became invalid. One reason for this behavior as mentioned by an expert is the long interval between the updates – this expert talked about maintaining them mostly once a year – requiring a lot of adaptions.

7.3.2 Requirement relevance

This subsection focuses on the defined requirements and their relevance to the experts. Question 14 of the questionnaire asked for the usefulness of the given features in POGito, which were based on the defined requirements. Each bar in figure 7.3 represents the mean of the results for the mentioned question. They can also be found above the bar in textual form. For each question, a number between 1 (not useful) and 4 (very useful) was requested from the interviewee. It should be noted that even though the prototype might not support the programming language or page objects of the experts, the interviewee was asked to answer the questions based on the relevance for them in general.

As visualized in figure 7.3, almost all features – and therefore also the defined requirements – are considered relevant for a tool supporting the page object maintenance. Except one question every other one was rated in their relevance with at least 3 out of 4 points on average.

Question 14.5, which focused on the requirement R8 "Displaying all found variables on a page", was rated to be the least relevant one with a mean of 2,2. One expert, for example, mentioned the missing relation from the variable to the highlighted element on the web page. Having many elements in a page object, this would result in a massive amount of highlighted elements, which would give no information to the user anymore. However, the experts saw a potential in the feature when adapted, e.g. when providing an additional textual information to the object.

The second lowest-rated question was Q14.7 – with requirement R3 as base – which focused on the usage of multiple page objects at the same time in the tool. The mean rating of this question was 3 out of 4. Even though this represents a good result in the rating scale, it is low in contrast to the other ratings. One comment on this requirement was the missing link between the page object and the browser state – requiring to know which page object expects which web page. Based on this problem, switching page objects in the tool also requires the effort of knowing which state is required and how to get there. The expert suggested putting the URL or state in the page object so that the tool can interpret it in order to switch the web page in the browser to the given state.

When asked for the most appealing feature of POGito – which was done in question 15 – the opinion of the experts split. There was only one feature, the verification of variables, which was chosen twice. The other features chosen were import/export, easy selection of a path and displaying of all variables on the page. It should be noted, that even though two experts chose the same feature, this can also be seen as a coincidence. To receive a more accurate result for this question, more experts would be needed.

Overall, all defined requirements were mentioned to be relevant and important to the experts, hence they can be seen as confirmed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

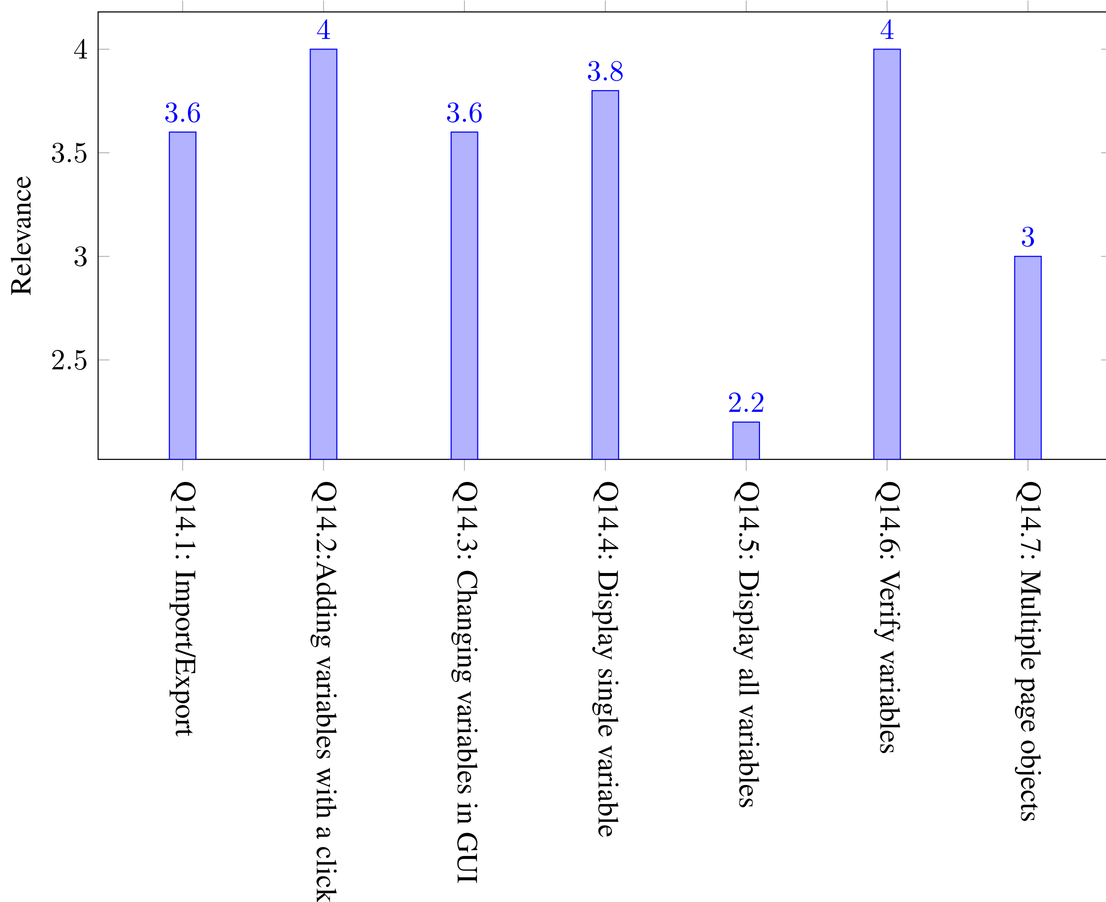


Figure 7.3: Relevance for the given feature between 1 (not relevant) and 4 (very relevant)

7.3.3 Feedback on the prototype and its functionality

In contrast to the previous section where features were rated based on a scale, this section will discuss the general opinion of the experts on the prototype, its benefits and drawbacks and whether or not the tool for page object maintenance would help in their daily work.

One main goal of this thesis is to enhance the current situation testers face when using page objects. When the experts were asked whether or not the prototype would enhance their daily routines, 4 out of 5 experts answered "yes". The one expert answering "no" on this question pointed out that the current concept how the tool is handling page objects is not yet ready for use in their project, however, the general idea is interesting. This expert further mentioned that the chosen approach of POGito to export the paths using the annotations is too static and would cause problems in their project. On the other side, one expert mentioned that every tool which supports the maintenance is a great help in contrast to manually editing files. Finally, it should be mentioned that, when some experts answered yes, they referred to the general concept of the tool and not to the support for their current project.

Even though the experts found the prototype helpful in their daily routines, only 2 experts would see the prototype as solving their problems mentioned before. The main reasons for this are missing features essential for them in order to replace current approaches. The feature missed most is explicit support for object inheritance or page objects defining only a subset of a page. Further missing features are the lack of support in programming languages, other page object styles and mobile page objects for experts working on mobile applications. The tool was seen as a good starting point for new projects or new developers without much knowledge in the field, but

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

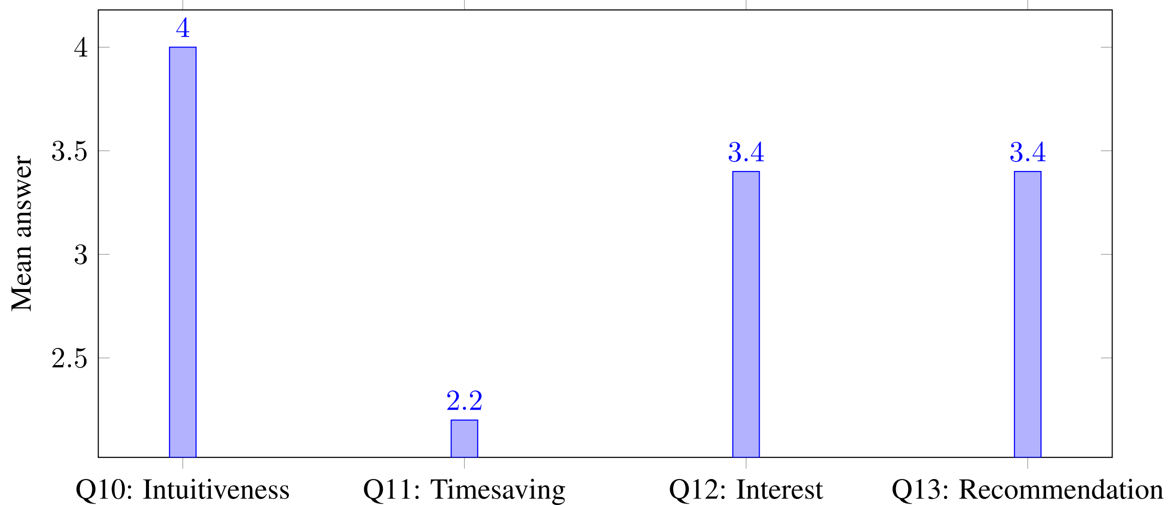


Figure 7.4: The mean value of the answers given by the experts on the defined question from 1 (most negative option) to 4 (most positive option) when asked about POGito

to have too little functionality to support advanced projects. Some experts also did not like the many steps required to interact with the application when using the import and export features or validation and would prefer a more automated, integrated solution for their project either via the IDE or via source code management. When asked if the tool could handle the experts' project regarding the size and complexity of their page objects, another downside mentioned was the manual navigation in the page. One expert, for example, mentioned that they often do not know which page objects are used for this specific page and state and that a manual interaction in the browser would be very hard. Consequently, an automatic state change when changing between page objects would be preferred. The last question regarding the enhancement for the experts was question 11, which asked on a scale from 1 (no time saved) to 4 (much time saved), how much time the expert would save in their daily work using the prototype instead of current approaches. As shown in the second bar from the left in figure 7.4, the mean was 2.2, meaning that not much time would be saved. This low number resulted from the previously mentioned fact that for most experts the tool would not be usable in their current setting due to the prototypical state. However, the number also shows that, though little, some time can be saved with the chosen approach even in the current state and that an adaption to the project's requirement would enhance it even further.

Question 10 tried to evaluate the user interface and concept of POGito in order to also find downsides in the developed prototype itself without the background of requirements. This question, therefore, asked the experts how intuitive POGito is seen by them on a scale from 1 (not intuitive) to 4 (very intuitive). The mean of the result can again be seen in figure 7.4. The mean of 4 clearly shows that all experts found the tool very intuitive and liked the way the user interface was designed and how the user could interact with it.

In order to evaluate the general interest on a tool like the one developed in this thesis, the interviewees were asked for that in question Q12 and their likelihood for endorsements of the prototype to other colleagues in the field in question Q13. The means of these two questions can also be found in figure 7.4 and are for both answers 3,4. However, these means might imply to both be equally answered. While the first question, asking for the interest in integrating the prototype in their daily routines, only contained answers with 3 and 4, the second question, asking for a recommendation to other colleagues, also included a rating of 2. This rating of 2 originated due to the fact that the prototype is for the expert not yet ready for further usage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Overall, the presented prototype went down very well on all experts and received positive feedback. As the prototype was built based on the defined requirements, those can be seen as approved and relevant for the experts, whereas some might be more relevant in their daily work than others. Especially for young software testers, the prototype would be a benefit. However, the prototype is not yet considered ready for production. Adaptions required on the prototype as mentioned by the experts are a better integration to the project, wider support of programming languages and page object styles. These mentioned key factors for the experts can be seen as future work on this topic.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

8 Conclusion

This chapter concludes this thesis by giving a short summary of each chapter along with a recap of the answered research questions in section 8.1. Following on that, section 8.2 will then recommend future work topics based on the conducted evaluation.

8.1 Recap

The thesis started with an introduction to a problem software testers face when using the page object pattern in their system level tests. The main drawback of the page object pattern is that even though page objects should enhance the maintainability of software tests by providing a layer between test and the system under test, it often requires a lot of additional effort to maintain them due to very static element identification. In order to solve this problem, chapter 1 defined three research questions, which were answered throughout this thesis and based on which the current situation should be enhanced.

Chapter 2 emphasized the relevance for software testing and the automation of it and provided further information on the field. This information included different test levels and test types along with general knowledge required to understand the chapters following on that. The chapter ended by explaining user interface testing, identification methods for elements in the user interface and by describing the page object pattern, a pattern commonly used in model-based integration testing.

The first research question was answered in chapter 3 when different available solutions were discussed. While many solutions could be found during the research, none of the tools provided a maintainability aspect like reusing existing page objects. However, some of the tools already included functionality enhancing the general workflow of a software tester.

Based on this functionality, chapter 4 then answered the second research question by defining requirements to a tool which enhances the current situation of software testers with respect to maintainability. Before these requirements were defined, the term maintainability and its relevance for a software developer was also discussed in this chapter.

Using this list of requirements and further relevant functionality from the state of the art tools, a prototype was developed. The development was described both in chapter 5, where the basic concept of the tool was defined along with user interface mock-ups and in chapter 6, where the implementation process of the prototype called POGito was described and code-samples provided for better understanding. As a result, a browser extension for Chrome was developed, which tester could import and export their page objects, validate their paths and modify their variables with.

Chapter 7 then answered the third research question by conducting an expert interview with software testers. The process of the expert interview was explained in this chapter along with information about the conducted evaluation and the threats to validity which might have an effect on the result of it. Finally, the results of the evaluation were presented. These results were overall very positive and all of the experts liked the idea of the tool. A general benefit of the prototype was seen especially in a reduction of time resources while increasing the maintenance aspects of page objects. The prototype was regarded to heavily support testers with little experience in the field, but also to support experienced testers in certain tasks. However, also some drawbacks and wishes were mentioned by the interviewee which can be seen as future work, either on the defined requirements or the implemented prototype.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

8.2 Future work

The following subsections describe and propose future work on the implemented prototype based on the experts' feedback during the interview or found to be relevant while writing this thesis. It should be noted that some future work proposals might stand in conflict with other ones, as different project structures might have different additional requirements to the tool.

8.2.1 Better source code integration

One drawback for the experts was the manual interaction between POGito and the source code when maintaining it. Future work on the tool should focus on a better integration with the page object files. This could be done, for example, by moving the browser extension to an IDE plugin, as already discussed in chapter 5 but discarded due to time reasons. This integration to the IDE could also include a functionality mentioned by an expert: continuous evaluation. When working on the project and using the plugin, the paths of the page objects could be evaluated in the IDE throughout the process automatically. This would allow for easily detecting wrong paths while working on or executing tests.

Another approach which could support a better integration would be to use a project file and provide additional information – like URL or comments to the page object – in POGito. This project file could be synced via a version control system and reused by every developer in the team, allowing additional convenience when using the tool. By doing so, the browser could navigate to the correct page automatically, removing or at least reducing manual interaction with the web browser.

8.2.2 Wider range of programming languages

The current prototype only supports Java as a language for import and export. While Selenium is widely seen as the de-facto standard by the experts, different programming languages are used in production and required in daily life. Especially JavaScript, due to frameworks like Angular or React, should be supported by the tool.

8.2.3 Method maintenance

In the current implementation of POGito, methods are not part of the import and export of page objects. This support, in addition to validation or adaption of the variables in the methods, or even method support in the tool would be a massive improvement for experts, as mistakes can happen when merging the old page object with the new, maintained, one.

8.2.4 Support for different page object formats

A driving factor for a tool supporting the maintenance process of page objects is that the page object can be parsed by the tool. As the interview has shown, different page object styles are used in projects, resulting in a no-go for the tool when the project's page object style is not supported. Some of the most important style cases which are currently not supported by POGito but relevant based on the answers of the experts are the following:

- **Page object inheritance:** Even though POGito implicitly supports the inheritance of page objects – the user simply imports both files and only works on a subset of it – better support for object inheritance can be seen as future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **In-line and dynamic paths:** Some experts talked about their page objects mainly using in-line paths due to various reasons. Therefore, support for page objects using in-line or even dynamic paths should be evaluated for the tool.
- **Annotations:** One expert mentioned custom annotations used in his page objects as well. In order to preserve the content of page objects, the tool should at least reuse these annotations in the export.
- **Comments:** Similar to annotations, custom comments in page objects should also be included in the export after modification or supported to be maintained by the tool as well.

Taking the different programming languages into account, the style might also need to be evaluated and adapted to these programming languages, which might describe page objects completely differently.

8.2.5 Mobile support

Some experts mentioned page objects not only in different programming languages but also in other contexts like mobile applications. While hybrid apps could potentially be evaluated in the prototype, the aspect of native mobile applications has not been taken into account when designing the tool. Future work on this topic can also focus on the mobile part of page objects and should evaluate how these can best be supported by a tool like POGito.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

References

[6] David Arthur and Sergei Vassilvitskii. „K-Means++: The Advantages of Careful Seeding“. In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’07. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, 1027–1035. ISBN: 9780898716245.

[7] Paul Baker et al. *Model-Driven Testing: Using the UML Testing Profile*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 3540725628.

[9] International Software Testing Qualifications Board. *Certified Tester - Foundation Level Syllabus*. 2018.

[15] J. Collofello and K. Vehathiri. „An environment for training computer science students on software testing“. In: *Proceedings Frontiers in Education 35th Annual Conference*. 2005, T3E–6. DOI: 10.1109/FIE.2005.1611937.

[16] A. Contan, C. Dehelean, and L. Miclea. „Test automation pyramid from theory to practice“. In: *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*. Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 1–5. DOI: 10.1109/AQTR.2018.8402699. URL: https://doi.ieeecomputersociety.org/10.1109/AQTR.2018.8402699.

[21] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-43287-0.

[22] Michael Felderer et al. „Model-Based Security Testing: A Taxonomy and Systematic Classification“. In: *Softw. Test. Verif. Reliab.* 26.2 (Mar. 2016), 119–148. ISSN: 0960-0833. DOI: 10.1002/stvr.1580. URL: https://doi.org/10.1002/stvr.1580.

[23] Boni Garcia. *Mastering Software Testing with JUnit 5: Comprehensive Guide to Develop High Quality Java Applications*. Packt Publishing, 2017. ISBN: 1787285731.

[26] Boris Gloger. „Scrum“. In: *Informatik-Spektrum* 33.2 (2010), pp. 195–200. ISSN: 1432-122X. DOI: 10.1007/s00287-010-0426-6. URL: https://doi.org/10.1007/s00287-010-0426-6.

[27] Dorothy Graham and Mark Fewster. *Experiences of Test Automation: Case Studies of Software Test Automation*. 1st. Addison-Wesley Professional, 2012. ISBN: 0321754069, 9780321754066.

[28] Dorothy Graham et al. *Foundations of Software Testing: ISTQB Certification*. Intl Thomson Business Pr, 2008. ISBN: 9781844803552, 9781844809899.

[29] Thomas Grechenig. *Softwaretechnik: mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, 2010.

[30] Maxim Gromov et al. „Model Based JUnit Testing“. In: June 2019, pp. 139–142. DOI: 10.1109/EDM.2019.8823472.

[32] Dirk W. Hoffmann. *Software-Qualität*. 2nd ed. eXamen.press. Springer Vieweg, 2013. ISBN: 978-3-642-35699-5. DOI: 10.1007/978-3-642-35700-8.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[33] „IEEE Standard for Automatic Test Markup Language (ATML) Unit Under Test (UUT) Description". In: *IEEE Std 1671.3-2017 (Revision of IEEE Std 1671.3-2007)* (2018), pp. 1–104. ISSN: null. DOI: 10.1109/IEEESTD.2018.8337144.

[34] „IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[36] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[37] D. Janzen and H. Saiedian. „Test-driven development concepts, taxonomy, and future direction". In: *Computer* 38.9 (2005), pp. 43–50. DOI: 10.1109/MC.2005.314.

[41] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley, 1990. ISBN: 978-0-47031680-1.

[42] Barbara A. Kitchenham. „Systematic Review in Software Engineering: Where We Are and Where We Should Be Going". In: *Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies*. EAST '12. Lund, Sweden: Association for Computing Machinery, 2012, 1–2. ISBN: 9781450315098. DOI: 10.1145/2372233.2372235. URL: https://doi.org/10.1145/2372233.2372235.

[43] R. Lagerstedt. „Using automated tests for communicating and verifying non-functional requirements". In: *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*. 2014, pp. 26–28.

[44] M. Leotta et al. „Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013, pp. 108–113. DOI: 10.1109/ICSTW.2013.19.

[45] Peter Liggesmeyer. *Software-Qualität - Testen, Analysieren und Verifizieren von Software (2. Aufl.)* Spektrum Akademischer Verlag, 2009. ISBN: 978-3-8274-2056-5. DOI: 10.1007/978-3-8274-2203-3. URL: https://doi.org/10.1007/978-3-8274-2203-3.

[46] Johannes Link. *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*. 2nd ed. Heidelberg: dpunkt, 2005. ISBN: 978-3-89864-325-2.

[47] John J. Marciniak. *Encyclopedia of Software Engineering*. 2nd. USA: John Wiley & Sons, Inc., 2002. ISBN: 0471210080.

[48] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670, 9780735619678.

[51] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. USA: John Wiley & Sons, Inc., 2004. ISBN: 0471469122.

[55] M. Pezzè and M. Young. *Software testen und analysieren: Prozesse, Prinzipien und Techniken*. Oldenbourg, 2009. ISBN: 9783486585216.

[56] B. Potter and G. McGraw. „Software security testing". In: *IEEE Security Privacy* 2.5 (2004), pp. 81–85. DOI: 10.1109/MSP.2004.84.

[57] Rudolf Ramler and Klaus Wolfmaier. „Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost." In: Jan. 2006, pp. 85–91. DOI: 10.1145/1138929.1138946.

[59] T. Roßner. *Basiswissen modellbasierter Test*. dpunkt-Verlag, 2010. ISBN: 9783898645898.

[60] Alexander Schatten et al. *Best Practice Software-Engineering*. Heidelberg, Germany: Spektrum Akademischer Verlag Heidelberg, 2010. ISBN: 978-3-8274-2486-0.

[71] K. Sneha and G. M. Malle. „Research on software testing techniques and software automation testing tools". In: *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*. 2017, pp. 77–81. DOI: 10.1109/ICECDS.2017.8389562.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[72] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, 2019. ISBN: 978-3-86490-583-4.

[73] Andrea Stocco et al. „APOGEN: automatic page object generator for web testing“. In: *Software Quality Journal* (Aug. 2016). DOI: 10.1007/s11219-016-9331-9.

[74] Andrea Stocco et al. „Clustering-Aided Page Object Generation for Web Testing“. In: June 2016. DOI: 10.1007/978-3-319-38791-8_8.

[77] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN: 0123725011.

[80] M. Wahid and A. Almalaise. „JUnit framework: An interactive approach for basic unit testing learning in Software Engineering“. In: *2011 3rd International Congress on Engineering Education (ICEED)*. 2011, pp. 159–164. DOI: 10.1109/ICEED.2011.6235381.

[81] Dirk Wallerstorfer. „Improving maintainability with Scrum“. MA thesis. Austria: TU Vienna, 2011.

[82] Y. Wang, J. Yao, and X. Yu. „Information Security Protection in Software Testing“. In: *2018 14th International Conference on Computational Intelligence and Security (CIS)*. 2018, pp. 449–452. DOI: 10.1109/CIS2018.2018.00106.

[89] Dr. Pankaj Yadav, Umesh K. Yadav, and Surbhi Verma. „Software Testing : Approach to Identify Software Bugs *“. In: 2012.

[90] B. Yu, L. Ma, and C. Zhang. „Incremental Web Application Testing Using Page Object“. In: *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. 2015, pp. 1–6. DOI: 10.1109/HotWeb.2015.14.

[91] Markus Zoffi. „Entwurf und Implementierung eines GUI basierten RCP Frameworks zur Spezifikation und Modellierung von Testfällen zur Optimierung der White-Box-Komplexität bei hoher Software-Volatilität“. MA thesis. Austria: TU Vienna, 2014.

Online References

[1] *Android Espresso*. URL: https://developer.android.com/training/testing/espresso (visited on 01/26/2020).

[2] *Annotations in Java Documentation*. URL: https://docs.oracle.com/javase/tutorial/java/annotations/index.html (visited on 04/01/2020).

[3] *APOGEN*. URL: http://sepl.dibris.unige.it/APOGEN.php (visited on 01/26/2020).

[4] *APOGEN GitHub*. URL: https://github.com/tsigalko18/apogen (visited on 01/26/2020).

[5] *Appium*. URL: http://appium.io/ (visited on 11/29/2019).

[8] *Bamboo*. URL: https://www.atlassian.com/software/bamboo (visited on 11/29/2019).

[10] *Browser Extension Community Group*. URL: https://browserext.github.io/ (visited on 05/02/2020).

[11] *Browser Extensions Mozilla*. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions (visited on 05/02/2020).

[12] *Chrome APIs*. URL: https://developer.chrome.com/apps/api_index (visited on 05/02/2020).

[13] *Chrome extensions*. URL: https://developer.chrome.com/extensions (visited on 05/02/2020).

[14] *Chrome incompatibilities Mozilla*. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Chrome_incompatibilities (visited on 05/02/2020).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[17] *Content Scripts Chrome extensions*. URL: https://developer.chrome.com/extensions/content_scripts (visited on 05/02/2020).

[18] *CSS Selectors*. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors (visited on 03/28/2020).

[19] *Desktop Browser Market Share April 2020*. URL: https://netmarketshare.com/browser-market-share.aspx (visited on 05/30/2020).

[20] Quick Miriam Doughty-White Pearl. *Million Lines of Code*. URL: https://informationisbeautiful.net/visualizations/million-lines-of-code/ (visited on 11/29/2019).

[24] *Getting Started Tutorial Chrome extension*. URL: https://developer.chrome.com/extensions/getstarted (visited on 05/02/2020).

[25] *GitLab CI*. URL: https://about.gitlab.com/product/continuous-integration/ (visited on 11/29/2019).

[31] *Handlebars*. URL: https://handlebarsjs.com/ (visited on 05/12/2020).

[35] *IntelliJ Compare with clipboard*. URL: https://blog.jetbrains.com/phpstorm/2013/02/comparing-files-and-folders-within-your-ide/ (visited on 04/26/2020).

[38] *Jenkins*. URL: https://jenkins.io/ (visited on 11/29/2019).

[39] *jUnit*. URL: https://junit.org/ (visited on 11/29/2019).

[40] *jUnit Assertions*. URL: https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html (visited on 11/29/2019).

[49] *Message Passing Chrome extensions*. URL: https://developer.chrome.com/extensions/messaging (visited on 05/02/2020).

[50] *Mustache*. URL: https://mustache.github.io/ (visited on 05/12/2020).

[52] *New flaw discovered on Boeing 737 Max, sources say*. URL: https://edition.cnn.com/2019/06/26/politics/boeing-737-max-flaw/index.html (visited on 11/29/2019).

[53] *Page Modeller*. URL: https://github.com/danhumphrey/page-modeller (visited on 01/30/2020).

[54] *PageObject-IO*. URL: http://pageobject.io/ (visited on 01/26/2020).

[58] *React*. URL: https://reactjs.org/ (visited on 05/12/2020).

[61] *Selenium*. URL: https://selenium.dev/ (visited on 11/29/2019).

[62] *Selenium Code Generator*. URL: https://github.com/naukri-engineering/SeleniumCodeGenerator (visited on 01/26/2020).

[63] *Selenium Grid*. URL: https://selenium.dev/documentation/en/grid/ (visited on 11/29/2019).

[64] *Selenium IDE*. URL: https://github.com/SeleniumHQ/selenium-ide (visited on 11/29/2019).

[65] *Selenium IDE Github*. URL: https://github.com/SeleniumHQ/selenium-ide (visited on 05/12/2020).

[66] *Selenium JavaDoc WebElement*. URL: https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/WebElement.html (visited on 03/31/2020).

[67] *Selenium Page Object Generator*. URL: https://chrome.google.com/webstore/detail/selenium-page-object-gene/epgmnmcjdhapiojbohkkemlfkegmbebb (visited on 01/26/2020).

[68] *Selenium Page Object Generator GitHub*. URL: https://github.com/rickypc/selenium-page-object-generator (visited on 01/26/2020).

[69] *Selenium WebDriver*. URL: https://selenium.dev/documentation/en/webdriver/ (visited on 02/07/2020).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[70] *Selenium WebDriver Elementor Toolkit*. URL: https://github.com/sergueik/SWET (visited on 01/26/2020).

[75] *TestNG*. URL: https://github.com/cbeust/testng (visited on 11/29/2019).

[76] *TIOBE*. URL: https://www.tiobe.com/tiobe-index/ (visited on 04/07/2020).

[78] *W3*. URL: https://www.w3.org/TR/2015/REC-dom-20151119/ (visited on 01/26/2020).

[79] *W3Schools XPath Nodes*. URL: https://www.w3schools.com/xml/xpath_nodes.asp (visited on 10/28/2019).

[83] *WebExtension browser API Polyfill*. URL: https://github.com/mozilla/webextension-polyfill (visited on 05/02/2020).

[84] *WebExtension browser polyfill sendMessage*. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime/sendMessage (visited on 05/02/2020).

[85] *WTF PageObject Utility*. URL: https://github.com/wiredrive/wtframework (visited on 01/26/2020).

[86] *XML Path Language (XPath) 3.0*. URL: https://www.w3.org/TR/xpath-30/ (visited on 10/28/2019).

[87] *XPath and CSS selector comparison*. URL: https://johnresig.com/blog/xpath-css-selectors/ (visited on 03/28/2020).

[88] *Xpath cheatsheet*. URL: https://devhints.io/xpath (visited on 10/28/2019).

TU Bibliothek. WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

A Appendix

A.1 Interview

Interview #__ Name: ___________________________

Internal Notes:

Before the demonstration:

1) Please give an overview of your daily work. What is your job position? Which test-types are most used by you? How are you dealing with page objects in your tests? Which development process (agile, waterfall…) is most used when you are testing?

2) On a scale from 1 to 4 - where 1 is no experience and 4 is expert level-, how would you rate your experience with page objects?

   1 2 3 4

3) Did you encounter any problems using page objects yet?
   - No
   - Yes → 3.1) Which problems did you encounter?

4) How are you currently maintaining your page objects?

5) Do you currently use any tools for maintaining page objects?
   - No
   - Yes → 3.1) Which?

6) On an average project you are working on, how often do you have to…
   6.1) … maintain (update) a small subset of your page objects?
   6.2) … maintain (update) almost every page object of the project?
   6.3) … replace page objects completely?

Page 1/3

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Interview #__ Name: ___________________________

After the demonstration:

7) Based on the demonstration, do you think the provided prototype can enhance your daily work using page objects?
   - Yes
   - No → 7.1) Why not?

8) (If Yes on question 3): Based on the demonstration, do you think the provided prototype can solve the problems you mentioned?
   - Yes
   - No → 8.1) Why not?

9) Based on the demonstration, do you expect that POGito can handle all your page objects of a project and their variables in a convenient and clear way?
   - Yes
   - No → 9.1) Why not?

10) On a scale from 1 to 4 - where 1 is very hard and 4 is very easy-, how intuitive did you find POGito?

    1 2 3 4

11) On a scale from 1 to 4 - where 1 is no time saving and 4 is full-time saved -, how much time of your daily routine when working with page objects could be saved when working with POGito instead of your current approaches.

    1 2 3 4

12) On a scale from 1 to 4 - where 1 is no interest and 4 is total interest -, how much would you like to use POGito in your daily work with page objects?

    1 2 3 4

13) On a scale from 1 to 4 - where 1 is no suggestion and 4 very likely recommendation-, how likely would you recommend POGito to your colleagues?

    1 2 3 4

14) Please rate the usefulness of the given feature from 1 to 4 (1 means not useful, 4 extremely useful):
   - 14.1) Import/Export of page objects

    1 2 3 4

Page 2/3

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Interview #__ Name: ________________________

- 14.2) Adding new variables through a GUI based approach

  1 2 3 4

- 14.3) Changing or removing variables through a GUI based approach

  1 2 3 4

- 14.4) Displaying a single variable on the page, or giving information about its existence

  1 2 3 4

- 14.5) Displaying all found variables on the page

  1 2 3 4

- 14.6) Verifying all variables of a page object for their existence

  1 2 3 4

- 14.7) Support for maintaining multiple page objects at the same time

  1 2 3 4

15) Which feature of POGito did you find the most appealing?

16) Are there any shortcomings in the implementation which you expect to be included in a maintenance tool?
   - No
   - Yes → 16.1) Which?

17) Is there anything you would like to add?
   - No
   - Yes -> 17.1) What?

Page 3/3

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A.2 Scenario-Outline

Background: I am a Developer new to the team. I was given a test case that navigates to the address site in bugstore and changes it there. The test case is implemented using JUnit and uses page objects for modeling each page in the application. However, this test case is currently not working due to problems in the page object implementation.

Before I start with my task I make myself familiar with the Bug Store application.

POGito Use scenarios:

Scenario Number	1
Short Description	I want to add a missing element to the existing page object
Current situation	• The test stops on the start page as an element is missing
Expected Outcome	• I have added the required variable to the page object • The test runs and navigates to the login page
Step definition	1) I open POGito 2) On the startup page, I select “Using an existing page object” and select the malfunctional page object in the file chooser 3) I open a browser session in POGito and navigate to the page 4) I click on ‘Show all variables’ on the page 5) I find out that the login button is not covered by the page object 6) I add the menu button to the page object and name it loginButton 7) I export the page objects and fix their methods in GIT 8) I run the test to make sure the test navigates to the login page

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Scenario Number	2
Short Description	I want to create a page object for the login page with variables for the username-field, password-field and for the login button
Current situation	● The test stops on the login page ● I do not have a login page modeled as page object
Expected Outcome	● I do have a login page modeled containing the required fields ● The test proceeds and navigates to the start page in the logged-in state
Step definition	1) I create a new page object and name it 'LoginPage' 2) I navigate to the login page in the browser 3) I add a variable for username via select of the element and rename it username 4) I add a variable for password via select of the element and rename it password 5) I add a variable for login-button manually by entering the information and name it loginButton 6) I export both files to the test-suite and fix their methods in GIT 7) I run the tests and check that it is not failing on the login page anymore

Scenario Number	3
Short Description	I want to check if there are any invalid paths on the page and adapt them
Current situation	● The test fails on the start page in the logged-in state ● I do not know, if and which of the given paths on the page object are invalid
Expected Outcome	● The invalid path of the page object is corrected ● The test runs and succeeds
Step definition	1) In POGito I open the page object file for the start page in the logged-in stage 2) I log myself in to the application in the browser and navigate to the start page 3) I click on verify variables 4) I can see one entry marked as invalid 5) I click on the invalid entry, us the button "Select target" and choose the correct target on the page 6) I export the given page object back to the original place 7) I adapt the page objects methods via GIT 8) I run the test and check that it succeeds

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Scenario Number	4
Short Description	I want to adapt a path on the start page
Current situation	• The test runs successfully • I am not satisfied with the selected path of the menu item
Expected Outcome	• I have adapted the path of the menu item • The test still runs
Step definition	1) I check that I am still on the main page in logged-in state in POGito as well as in the browser 2a) I change the path by using the dropdown menu OR 2b) I change the path manually 3) I check if the path is valid by finding the element on the page 4) I export the page objects and fix their methods in GIT 5) I run the test to make sure the test still succeeds

Scenario Number	5
Short Description	I want to check if all variables are required on the start page
Current situation	• The test runs • Sonar gives me a message that two variables are unused • I am not sure if I still need these variables
Expected Outcome	• I have found out that I do not need both variables in my test • I have removed the variable from the page object • The test still succeeds
Step definition	1) I check that I am still on the main page in a logged-in state in POGito as well as in the browser 2) I click on 'Show all variables' on the page 3) I find out that two variables are not needed by my tests 4) I remove one variable via POGito, as will not use it in the near future 5) I comment one variable out via POGito, as I expect to use it in the near future 6) I export the page objects and fix their methods in GIT 7) I run the test to make sure the test still succeeds

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub