FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Automated Acceptance Testing on macOS – A survey study focused on Avast Passwords for mac
Student:	David Mokoš
Supervisor:	MSc Felix Javier Acero Salazar
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

1] Survey the different technologies available for writing acceptance tests on macOS.
2] Select an appropriate test stack: GUI driving technology, testing framework, etc.
3] Prepare application for acceptance testing based on the selected test stack.
4] Implement a suite of acceptance tests to cover the most relevant requirements of Avast Passwords for mac.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 28, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Automated Acceptance Testing on macOS – A survey study focused on Avast Passwords for Mac

David Mokoš

Supervisor: MSc. Felix Javier Acero Salazar

May 14, 2018

Acknowledgements

First and foremost, I would like to thank my thesis supervisor, MSc. Felix Javier Acero Salazar, for his patient guidance, advice and encouragement he has provided throughout the whole time I have been working in Avast. I have been lucky to meet such a hardworking, inspirative and helpful person who cares so much about my work and who always keeps his word. His curiosity and positive mindset inspired me and helped me particularly when exploring new ideas. His careful revising contributed hugely to the creation of the thesis.

I take this opportunity to express gratitude to all of the Department faculty members for their help and support throughout my studies. I also thank my parents for the endless encouragement, support and attention. Last but not least, I am grateful to my partner who supported me and accepted my limited free time during this venture.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 14, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Mokoš, David. *Automated Acceptance Testing on macOS – A survey study focused on Avast Passwords for Mac.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

Software development requires adequate methods for quality assurance. One of those methods is acceptance testing. This thesis introduces the automated acceptance testing and its need in the software development process along with its common pitfalls. It summarizes the existing tools and technologies that could be used for automating acceptance tests on *macOS* as well as best practices for writing maintainable automated acceptance test suites. The gathered knowledge from the literature review is put into practice through the implementation of an automated acceptance test suite for *Avast Passwords for Mac*, a *macOS* application developed by Avast.

Keywords acceptance testing, test automation, GUI (Graphical User Interface) driving on *macOS*, automated tests maintainability

Abstrakt

Ve vývoji softwaru je pro zajištění kvality nezbytné použití vhodných metod. Mezi tyto metody patří akceptační testování. Tato práce popisuje výhody akceptačního testování v procesu vývoje softwaru a zároveň upozorňuje na jeho časté problémy. Práce také shrnuje dostupné nástroje a technologie, které jsou vhodné pro automatizaci akceptačních testů na *macOS*, a zároveň popisuje osvědčené způsoby pro vývoj snadno udržovatelných automatických akceptačních testů. Znalosti získané z literární rešerše jsou převedeny do praxe skrze implementaci sady automatických akceptačních testů pro *Avast Passwords for Mac*, aplikaci pro *macOS* od společnosti Avast.

Klíčová slova akceptační testování, automatizace testů, ovládání GUI (Graphical User Interface) na *macOS*, udržovatelnost automatických testů

Contents

List of Figures

List of Tables

Introduction

A key challenge faced by software industry is quality. To provide the highest quality software, testing is an essential part of any development process [1]. Acceptance testing, an essential part of software testing, is typically conducted at the end of the release cycle to guarantee that all functional requirements of the software are met [2]. Acceptance tests are usually conducted manually by QA engineers each time the software is released. This process is very tedious, prone to human error, time consuming and thus expensive for the developer team. This does not fit with the paradigm of agile development, that relies on quick feedback and short development cycles [3]. To ease and speed up this process, the automation of acceptance tests may be seen as a promising initiative. It can significantly reduce the time QA engineers spend on repetitive tasks and also makes the feedback loop shorter, which helps to find the defects sooner, when they are less expensive to fix [2, 4].

However even acceptance testing automation has its pitfalls. First, unless carefully designed, automated acceptance test suites can be very expensive to develop and maintain. Tests may become too brittle and overly coupled with the implementation details of the application under test, or in the worst scenario, tests may become unreliable and report false positive results [5]. A second important issue that arises when developing automated acceptance tests is tooling support. This issue is particularly prominent in *macOS*. In fact, the existing tooling support for the automation of native *macOS* applications is relatively poor and the existing documentation around it offers very basic information. All matters considered, the automation of native *macOS* applications seems to be a barely explored area in the academic and developer communities.

Avast is one of the largest security companies in the world [6] that places high emphasis on the quality of its software and is therefore constantly looking to improve its software development process. Quality Assurance (QA) is considered to be one of top priorities of the company. QA engineers in Avast spend a large portion of their time on manual testing. A particular stage in the

software development cycle where manual testing is used is during acceptance testing. Since acceptance tests need to be conducted every time a new release candidate is produced, QA engineers spend a lot of their time on repetitive tasks neglecting other tasks that could bring more value to the application.

In this context, the automation of the acceptance tests may help reduce the time QA engineers in Avast spend in manual testing and allow them to focus their efforts in more productive tasks.

The second part of this thesis will demonstrate the usage of automated acceptance tests applied to *Avast Passwords for Mac*, a password manager developed by Avast.

Goals of the Thesis

The aim of this thesis is to explore the possibilities available on *macOS* for building automated acceptance tests suites, to point out their potential benefits and to highlight their common problems. To validate the information gathered in the theory, this thesis will describe the development of an automated acceptance test suite that covers the most important use cases of *Avast Passwords for Mac*.

A secondary goal of this thesis is to contribute to the testing community by submitting improvements to open source projects such as *Appium For Mac*, as well as to provide a useful learning resource about common best practices for developing automated acceptance tests through implementation examples. In conclusion, this thesis should be a good source of information for QA engineers implementing an automated acceptance test suite for a *macOS* application.

Chapter 1

State of the Art

1.1 Software Testing Fundamentals

Testing is a fundamental tool for ensuring the quality of the software [7]. It can be characterized as a process of executing an application with the intent of finding errors [1]. Studies indicate that testing consumes more than 50% of the development time [8, 9]. On safety critical applications, testing can take up to 80% of the development time [7].

There are many types of tests. Each of these types focuses on a different aspect of the application and should be considered as part of the QA processes in the software development cycle. The diagram 1.1, presents various types of tests according to whether they are *business facing* or *technology facing* and whether they *support the development process* or they *critique the project*. [2]

The most common categories of tests along with their definitions are presented below.

1.1.1 Business Facing Tests that Support the Development Process

Functional Acceptance tests

The main focus of this thesis are functional acceptance tests that are fundamental part of this category. They will be discussed deeply in section 1.2.

1.1.2 Technology Facing Tests that Support the Development Process

There are three main types of tests in this category. All of them are developed and maintained consecutively by developers.

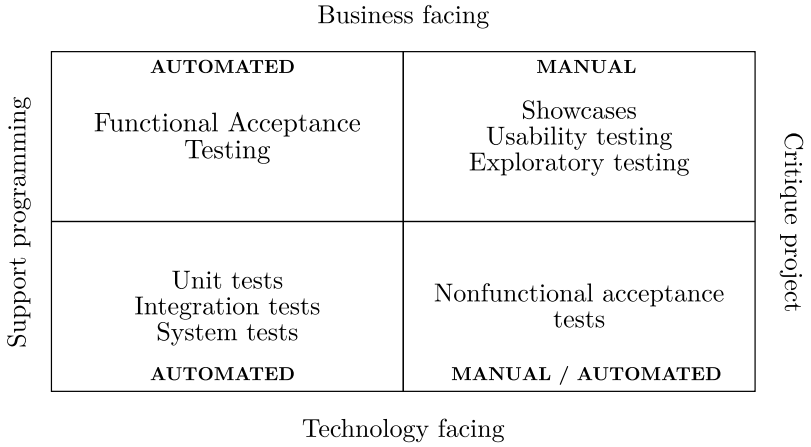

Figure 1.1: Brian Marick's four quadrant diagram describing types of tests used in software projects, adopted from [2, p. 85 (modified)]

Unit Tests

One of the main types of tests in this category are *Unit tests*. They occur at the lowest level of abstraction. For instance, they test a method or module in isolation from the logic and architecture of the application. To ensure maximum error detection, unit testing relies heavily on *white box testing* techniques, in which the tester has the knowledge of the internal structure of the program and uses it as an advantage for designing better tests [1]. They should not cover any interaction between parts of the application or interaction with the environment, such as a file system or an external service. [2, 7, 10, 11]

Component Tests

Component tests (sometimes called *Integration tests* [2]), on the other hand, are performed against groups of methods or modules to ensure that they work together as a whole. Component tests can catch errors caused by incorrect management of the life cycles of objects or data. To address these kinds of errors, they rely on *black box testing* techniques, in which the tester is completely unconcerned about the internal behavior and structure of the application. [1, 2, 7, 10, 11]

Deployment Tests

A different type of tests in this category are *Deployment tests*. They ensure that the application can be correctly installed, configured and run in the production environment – the environment where the application is supposed

to perform. Deployment tests also make sure that the application is able to communicate with all external services and is responsive. [2, 7]

1.1.3 Business Facing Tests that Critique the Project

There are three primary types of tests in this section, mostly performed manually. These are the tests conducted to verify that the application provides the expected value to the users. These tests not only check requirements, but also check whether the requirements are correct and a user would be satisfied with the experience of using the application [2].

Exploratory tests

Exploratory tests are particularly important. They can be described as "simultaneous learning, test design, and test execution" [12, p. 2]. In other words, "exploratory testing is any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests" [12, p. 2]. Exploratory testing is a creative process which leads not only to finding errors, but also to suggesting improvements or designing new tests based on the results and experience. [2]

Showcases

Other important addition to this category are *showcases*. They are frequently performed to demonstrate the customers the functionality the team have delivered. They should be realized as often as possible to provide the customer an opportunity for suggesting improvements or even radical changes in the application early in the development process when the changes are not as expensive. [2]

Usability Tests

Usability testing is another way of getting feedback about the application, its functionality and user experience. There are different approaches to usability testing. One of the common approaches is contextual inquiry – watching users while they work with the application to get the data from the observation. Another approach similar to the previous one is to give the users some scenarios and observe the users while they are trying to achieve them. There are some other methods, such as eye tracking or first click testing. [1, 2, 10]

In summary, these tests are conducted to collect data about the usage of the application. This data can help reveal whether the application is useful, comfortable to use, intuitive and consistent.

1.1.4 Technology Facing Tests that Critique the Project

Nonfunctional Acceptance tests

These tests are used for testing the non-functional requirements of the application. These are the qualities of the application other than its functionality. Non-functional requirements may include reliability, security, availability among others. The tests used to verify whether these criteria have been met are quite different from those verifying the functional requirements. These tests often need significant resources, such as special environments in which the application runs or special tools to simulate different behaviors of the system, such as limited network access or disk space. [2]

1.1.5 Regression Testing

Regression testing is a cross-cutting type of testing and thus is not mentioned in the four quadrant diagram in figure 1.1. *Regression testing* is applied after the software is modified during the development process to make sure that no new errors were introduced by the changes. If done properly, it can help ensure that fixing an error does not introduce any new errors in other parts of the application. [1, 10, 13]

1.2 Acceptance Testing

1.2.1 Overview

Acceptance testing can be defined as "testing conducted to determine whether a system satisfies its acceptance criteria and to enable the customer to determine whether to accept the system" [14, p. 22]. Acceptance testing is a testing method conducted at the very end of the development cycle that tests the overall functionality of the system to determine whether the software has met the acceptance criteria and whether it provides end users with expected functionality [2,10]. The acceptance criteria is the subset of software requirements that the application must fulfill in order to by accepted by a customer [15]. The main difference between *Acceptance Tests* and *Unit Tests* is that *Acceptance Tests* are business-facing, not developer-facing.

An acceptance test is a formal description of the behavior of an application, usually expressed by scenarios or examples. Different approaches have been proposed for writing these scenarios, oftentimes with the aim of having the possibility of automating the execution of the tests with the help of appropriate tools. [2, 16]

The whole team should be responsible for the *acceptance testing* process, including developers, QA engineers and business participants such as product owners. The product owner is responsible for writing the software requirements and deriving scenarios from them. Software developers should always

make sure that the requirements are met for all new functionalities. Software testers should go through the scenarios and verify the expected functionality of the software. This process needs to be done at least once before each release of the application, but is advisable to do it as frequently as possible to catch errors early in the development process when the cost of the fixing them is not so high. [2]

There are clear advantages that make *Acceptance Testing* an essential stage of the software development process. First and foremost, because they simulate the behavior of the end user and run on a production like environment, acceptance tests help identify problems that may be missed by other types of tests like *unit* or *component tests*. Another advantage of *Acceptance Tests* is that they capture the customer requirements in a way that can be directly verified and provide an unambiguous *contract* between the customer and the development team. In addition, since developers, QA engineers and customers (or a product owner) need to work closely on the *Acceptance Tests*, the collaboration between them is strengthened. This results in the whole team having the same focus: to make sure the application delivers the expected business value. [2, 10]

1.2.2 Automation

Acceptance test automation has always been a controversial topic. In some circles, automated acceptance tests are seen as too expensive to create and maintain [5]. In fact, when implemented poorly, the cost of maintaining an automated acceptance test suite can offset its benefits. On the other hand, and as suggested by the experience related by several case studies, the value of a properly designed automated acceptance test suite clearly exceeds its cost. In fact, the cost is much lower than that of manual *acceptance* and *regression testing* and even lower than that of releasing a poor-quality application. [2, 4, 17, 18]

In general, tests automation has some undoubted benefits.

Faster Tests Since automated tests are written in code, they can be executed several orders of magnitude faster than manual tests.

Optimized QA resources They reduce the time testers spend on repetitive tasks allowing them to concentrate on higher level tasks, such as *exploratory testing*.

Better Test Reliability Automated tests reduce the room for human errors which may occur when performing repetitive tasks. [2, 10]

In addition to these, automated acceptance tests have other benefits.

Increased Defect Detection They can catch problems that neither unit nor component tests could. For example, they can catch threading problems, architectural mistakes or configuration and environmental problems.

Timely Defect Detection Running automated acceptance test suite in the *Continuous Integration* pipeline makes the feedback loops much faster, allowing for defects to be found and fixed early when the cost is not so high.

Faster Developer Feedback Automated acceptance tests can also be a valuable tool for the developers. They provide them a quick response as to whether they have finished a particular task. Developers can easily run the automated acceptance test suite during development and see which requirements of the task have been met and which are yet to be implemented.

Safer Code Modifications When used by developers, the automated acceptance tests can ensure that a modification of some part of the application does not break any existing functionality, which can be particularly helpful when making large changes across the application code. [2,10,19]

Despite the aforementioned benefits, is important to highlight that automated acceptance tests **do not** replace other types of testing. Although, acceptance tests **do** verify the overall functionality and **are** able to catch some cross cutting errors, they cannot completely replace *unit* nor *component testing*. Similarly, automated acceptance tests *do not* eliminate personnel costs because without constant maintenance, the automated acceptance test suite could easily become unusable. [2,18]

1.2.3 Implementation Patterns

The biggest challenge faced by an automated acceptance test suite is the cost of maintainability. This cost can be offset by following proper and verified practices [2, 19]. Similarly, writing maintainable automated acceptance test suites requires the help of the appropriate tools [2, 10]. These practices and tools will be detailed in this section.

Behavior driven development

An important part of developing a maintainable automated acceptance test suite is the analysis process. During this process, the acceptance criteria are defined and are used as the starting point for deriving the acceptance tests. Consequently, failing to define the acceptance criteria, could lead to an acceptance test suite that does not deliver the expected value to the project stakeholders. Similarly, automating badly defined acceptance criteria would

contribute to a hard-to-maintain and thus very expensive acceptance test suite [2,10]. Acceptance criteria must therefore be defined very carefully, keeping in mind that they will be the starting point from which the automated acceptance test suite will be derived. [2, 20, 21]

There are several approaches that can be used for defining the acceptance criteria. For manual tests, IEEE Computer Society has released a *Standard for Software and System Test Documentation* [22] that offers a well documented template for defining test cases.

A common issue in long term software projects is that the acceptance criteria are written at the beginning of the development process and become outdated as soon as new functionality is added to the project [2]. Outdated acceptance criteria cause troubles while testing the functionality of the application, and should be addressed as soon as possible. An strategy that can help mitigate this issue, is to use human readable test suites, which connect the acceptance criteria directly to the tests implementation [23]. This way the acceptance criteria are kept alongside the test code and can be easily updated. There are tools and techniques that allow the usage of acceptance criteria as executable specifications. One of this approaches is known as *behavior driven development*.

Behavior driven development (BDD) is an agile software development technique that encourages close collaboration between QA engineers, developers and non-technical business participants in the project. It focuses on defining clear specifications that are easy to read and write, are unambiguous, and can be easily bound with implementation code. This approach has several benefits. First, by having the acceptance criteria as a part of the acceptance test suite, they are being kept up to date as new tests are developed. Second, because an automated acceptance test suite that uses a BDD approach uses natural language for defining the acceptance criteria, it can be easily written and updated by non-technical members of the project, like product managers. Third, having the acceptance criteria as part of the automated acceptance test suite avoids duplication of the requirements specifications and the acceptance tests. [2, 23]

In most cases, BDD uses natural language constructs for defining the acceptance criteria. Tools like *Cucumber* [24], *JBehave* [25] or *Behave* [26] use a domain specific language called *Gherkin*. *Gherkin* uses plain English language with some additional structure to describe even the most complicated scenarios. *Gherkin* is designed to be easy to learn even for non-technical personnel. [23, 27]

In *Gherkin*, the scenarios that describe the acceptance criteria are written in `.feature` files. The `.feature` file describes one functionality of the application by defining one or more scenarios which consist of a series of steps. Each step needs to begin with the keyword `Given` followed by `When`, `Then`, `And` or `But`. A sentence starting with `Given` puts the system into a known state that stands as a prerequisite for the scenario. `When` steps describe the

key actions that need to be performed in order to fulfill the scenario. Finally, `Then` steps verify the outcome of the scenario. `And` and `But` keywords serve to improve the readability of the steps. An example of a `.feature` file written in *Gherkin* language is shown in section 2.3. [23, 24, 27]

The tools discussed above allow to link the acceptance criteria with the implementation code that drives the application under test. They typically use a regular expression matching to parse the steps defined in a domain specific language, such as *Gherkin*. *Behave* [26], for instance, is a *Python* framework that binds *Python* code with the acceptance criteria defined using the *Gherkin* language. *JBehave* [25] is a *Java* framework that implements the same functionality. *Cucumber* [24] is a framework written in *Ruby* and offers integrations with more than dozen programming languages and frameworks.

In summary, the essence of *Behavior Driven Development* is making the acceptance criteria executable. This is a big improvement over alternative approaches like writing the acceptance criteria in an *Excel* sheet or a *Word* document.

Application Driver Layer

An issue that negatively impacts the maintainability of automated acceptance test suites is the tight coupling between the tests and the GUI of the application under test. This issue is amplified by most GUI test recording tools which, unless carefully utilized, produce test code that is overly coupled with the GUI of the application under test.

Tools, such as *Marathon* or *TestComplete* usually offer an automated test recording feature that makes writing the first version of the tests very easy [28–30]. However, these tests are overly coupled with the GUI of the application under test, and thus they tend to be very brittle. [2, 17, 18]

One way to address this problem is by adding an abstraction layer between the test implementation and the GUI of the application under test [2].

The *Application Driver Layer* is the layer that lays closer to the application under test. It understands how the application works and how to communicate with it. It provides a high level API for controlling the application. For instance, if the application under test has a GUI, the *application driver layer* would know about the buttons and text fields on the screen and would be able to perform operations such as clicking the buttons and writing text to the text fields. [2]

From the implementation perspective, there are several tools that can be used for driving the GUI of the application under test. Some of these tools, however, are platform and technology specific. In the case of *macOS*, there are several alternatives available. The section 1.3.2 present the most prominent tools for driving GUIs on *macOS*. Their benefits and disadvantages in the context of *acceptance testing* are highlighted in chapter 2.

Page Object Pattern

Keeping the *application driver layer* as decoupled as possible from the application under test is an important part of preserving the maintainability of the automated acceptance test suite. In this context, a pattern like the *Page Object* can also be very useful [31].

The *Page Object* pattern is used to model areas or whole screens of the application as individual objects. All the functionality of a single screen is encapsulated into the *Page Object* which in turn should manage all the interactions with the screen. Since all the interactions with the GUI of the application under test are implemented in a single place, this pattern reduces code duplication, facilitates modification, and makes the tests more resilient to changes on the application under test. [2, 31, 32]

Layered Architecture

In order to create maintainable automated acceptance test suite and to follow the implementation patterns mentioned above, a layered architecture is essential. Figure 1.2 shows a layered architecture of an automated acceptance test suite.

The *Acceptance Criteria* layer should follow the BDD principles and be written in a domain specific language. These criteria should be backed up by implementation code in the *Test Implementation Layer*. This layer, in turn, communicates with the *Application Driver Layer*, which interacts directly with the application under test.

1.3 Acceptance Testing on macOS

1.3.1 The macOS operating system

macOS is a UNIX operating system developed by Apple which powers every Mac [33]. It offers several security mechanisms including application sandboxing. A sandbox isolates applications from critical system components and enforces policies that constraint the behavior of an application. More important for our discussion, the sandboxing enforced by *macOS* prevents applications from controlling other applications or driving their GUI [34]. One way to overcome this limitation is to use the accessibility features offered by *macOS*.

Accessibility

macOS comes with a variety of assistive technologies that help people with disabilities to seamlessly use the *macOS* system and the applications running on it.

From the developer's perspective, enabling accessibility in an application is as simple as using the accessibility APIs offered by Apple. Using these APIs,

developers can include semantic data about the elements in the UI of their application. This data can include: the role of a specific widget (e.g: *button*, *checkbox*), a description, a label and a unique identifier. [35–37]

This semantic data can later be leveraged by assistive applications such as *VoiceOver*, a built-in screen reader, to allow disabled users interact with the application. Similarly, GUI driving tools can rely on this data to automate interactions with the elements in the GUI; hence the relevance of accessibility in the discussion of GUI driving tools in *macOS*.

1.3.2 GUI Driving Tools

Xcode User Interface Testing

Xcode is an integrated development environment for building applications for *macOS*, *iOS*, *watchOS*, and *tvOS* [38]. It was first released in 2003 and is available via the *Mac App Store* free of charge for *macOS High Sierra* and *macOS Sierra* users [39].

Xcode UI Testing is the most recent addition to the Apple test toolkit. It gives developers the ability to validate the UI requirements of their applications in an automated fashion. It includes a UI recording feature, which helps developers generate test code by simply interacting with the GUI of the application under test. There are also enhanced test reports that provide detailed information about the test execution, including snapshots of the state

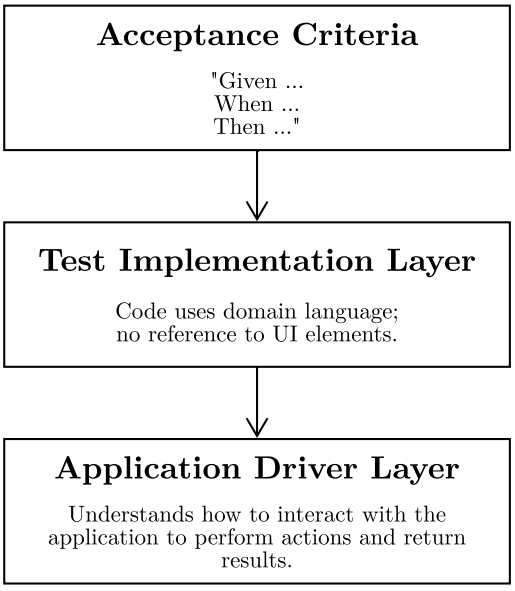

Figure 1.2: The three layers of maintainable automated acceptance tests, adopted from [2, p. 191 (modified)]

of the user interface at test failures. This tool can be used for both checking correctness and evaluating performance. [37, 40, 41]

Xcode UI Testing relies on two core technologies: the *XCTest* framework and the accessibility features of *macOS*, described in section 1.3.1 [37].

XCTest is a framework that enables a good part of the UI testing features integrated in *Xcode*. The framework contains base test classes as well as an API for assertions. Since XCTest is used for the creation of both unit and UI tests, the usage and creation of UI tests is very similar to the creation of unit tests. Finally, like most of the frameworks offered by Apple, the *XCTest* framework is fully compatible with both *Objective-C* and *Swift*. [37]

The API for writing UI tests is based on three classes: `XCUIApplication`, `XCUIElement` and `XCUIElementQuery`. `XCUIApplication` and `XCUIElement` classes represent the application under test and the UI elements in it. The `XCUIElementQuery` class is used for querying and finding those UI elements. To help developers with the structure of the tests, *Xcode* offers a feature for recording the basic steps of these tests. The generated test code can then be edited to, for instance, add assertions. [37]

AppleScript

AppleScript is a technology readily available in recent versions of the *macOS* operating system. On one hand, *AppleScript* is a scripting language that provides direct control over scriptable applications and over many parts of the *macOS* operating system [42]. On the other hand, *AppleScript* also describes the underlying technology supporting the language [43].

The automation offered by *AppleScript* is based on the *Open Scripting Architecture* [44], which provides a standard and extensible mechanism for interprocess communication in the *macOS* operating system. The communication occurs via messages called *Apple Events*, to which scriptable applications can respond to by performing operations or supplying the requested data [42, 45]. In this context, a scriptable application is an application that can recognize a variety of *Apple Events* that can be leveraged by an *AppleScript* [46].

While Apple has made it easier to add scriptability support to an application [45], the task still requires action from the developers, who need to write code to manage the different *Apple Events* sent to their applications by an *AppleScript* [43].

It is possible, however, to use *AppleScript* with applications without support for scriptability. To achieve this, an intermediate component called *System Events* is required. The *System Events* component acts as an intermediary between *AppleScript* and the application, enabling basic operations such as finding GUI elements, performing clicks and key presses on them as well as retrieving responses from these actions [45].

The actual scripting language offered by *AppleScript* is rich, object-oriented and capable of performing complicated programming tasks [47]. Its syntax

uses English-like constructs in an effort to be more accessible for less technical users. This being said, the language also has some limitations. For instance it lacks APIs for advanced numeric operations like logarithms or trigonometric functions [43]. Similarly, it features rudimentary text processing APIs with no support for the regular expressions. Some of these limitations, however, can be addressed through interactions with other scripting languages available in *macOS*, which include *Python*, *Ruby* and *Bash* [43, 45].

Javascript for Automation

Until recently, *AppleScript* was the only *Open Scripting Architecture component*[1] provided by Apple. However, one of the design goals of the *Open Scripting Architecture* was to enable integrations with other scripting languages [45]. With the release of *OS X 10.10 Yosemite*, *JavaScript for Automation* became a peer to *AppleScript*, thus allowing users to control scriptable applications using *JavaScript* [48].

Using *JavaScript* developers could create automation scripts in a familiar scripting language, enabling more complex use cases, while improving the readability and maintainability of their code; not to mention access to a more robust and extensible library of native functions. [48, 49]

Appium For Mac

Appium is an open-source tool for automating native, hybrid and web applications. Thanks to its cross-platform support, tests can be easily shared across platforms. [50]

Although the original *Appium* project only offers official support for *Android*, *iOS*, *Windows* and *FirefoxOS* [51], in 2013 the developers started *Appium For Mac*, an open-source project hosted on *GitHub* [52] for automating native *macOS* applications [53].

Appium For Mac implements the *WebDriver API* following the same approach as the *Chromedriver* or the *Firefox driver* for driving web browsers [54]. It is a standalone HTTP server that runs on the destination device (the device running the application under test) and relies on the *JSON Wire Protocol* [55] for communicating with the client (the device running the test suite). The *JSON Wire Protocol* is a transport protocol developed by *WebDriver developers* that implements a standardized set of endpoints exposed via a REST API [53, 56]. This client-server architecture opens interesting possibilities such as allowing the test code to be written in any programming language with support for the *WebDriver API* [54] thus effectively decoupling the test code from the implementation code of the application under test [51].

[1] Under the *Open Scripting Architecture*, the scripting language is implemented by a *component*. These components can be installed dynamically. [45]

The current implementation of *Appium For Mac* is built on top of a commercial, closed source, framework called *PFAssistive* [52]. This framework is responsible for the interactions with the GUI of *macOS* applications, and does so via the *Accessibility APIs* offered by Apple [57].

Chapter 2

Analysis

The state of the art introduced the QA benefits brought by automated acceptance tests and presented the tools available for their implementation in the *macOS* platform. This chapter describes the usage of these tools for implementing an acceptance test suite that covers the fundamental use cases of *Avast Passwords for Mac*.

2.1 The Application Under Test – *Avast Passwords for Mac*

2.1.1 Introduction

Avast Passwords for Mac is a password manager developed by Avast Software s.r.o. [58]. It is an application that securely stores all users' logins, notes and credit card details. It also offers a password generator that generates and saves a unique and strong password for each account. The user data is encrypted with the use of both symmetric and asymmetric cryptography and protected by a single *Master Password* [59].

Besides *macOS*, the *Avast Passwords* application is also available on *iOS*, *Windows* and *Android* and offers users the possibility of synchronizing their data across their devices.

Figure 2.1 shows one of the main screens of *Avast Passwords for Mac*–the credit card list screen.

2.1.2 Features

The most important features of *Avast Passwords for Mac* are:

Storing sensitive data Like the clients available for other platforms, the main functionality of *Avast Passwords for Mac* is storing delicate data, such as `logins`, `credit cards` and `notes`. All data is stored in a secure

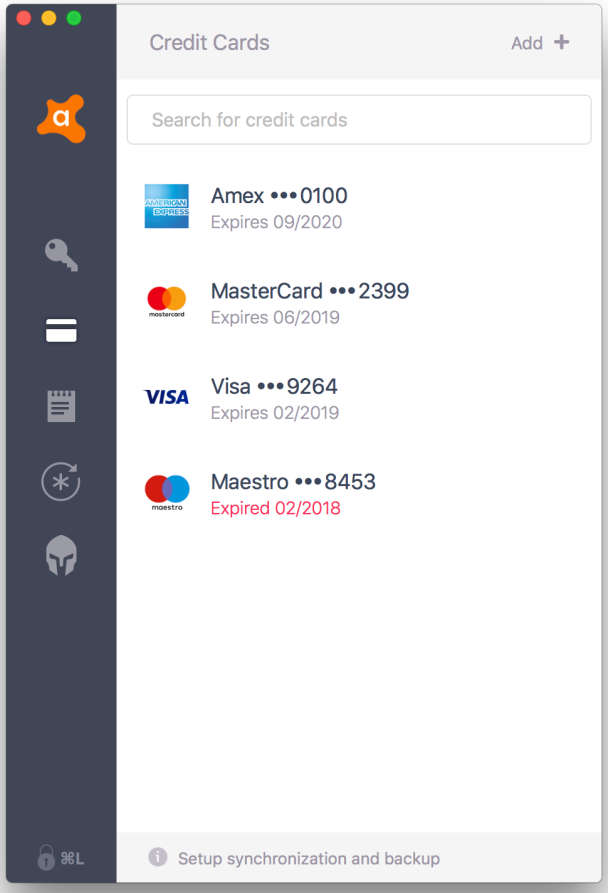

Figure 2.1: The credit card list screen of *Avast Passwords for Mac*

vault that is encrypted by a single *Master Password*. Users need to provide the *Master Password* for accessing the data. Additionally, users can set different triggers to lock their vault. For instance, users can choose to lock their vault automatically after certain period of inactivity, after their computer goes to sleep or right after they logout of their user session.

Synchronization across platforms Users can synchronize their data across supported devices and platforms with the aid of the Avast infrastructure. To enable the synchronization, users only need to create a free Avast account.

Communication with Browsers An important component of the application is the browser extension which is available for *Safari*, *Chrome* and *Firefox* and which allows users to automatically save and fill up the logins and credit card details on websites, as well as generate secure passwords for new accounts.

Password Generator Using the *Password Generator* users can create unique strong passwords to better protect their online accounts.

Password Guardian *Avast Passwords for Mac* also offers one paid feature – *Password Guardian*. The *Password Guardian* periodically checks against Avast servers for information about recent password leaks and advises users to change their password in case one of their accounts has been compromised. The *Password Guardian* also detects duplicate and weak passwords and notifies users about them.

2.1.3 Architecture

Avast Passwords for Mac requires the interaction of three components: the *Main Application*, the *Helper Application* and the *Browser Extension*. Each of these components offers assistance to the user in a subset of specific tasks. To the user, these components appear equally important, however, from a technical point of view, the *Helper Application* is the one that packs most of the business logic and is the only one that can access the encrypted user data. This design optimizes the usability of the application while focusing the code that has access to the protected data in a single point.

The different components of *Avast Passwords for Mac* as well as the main interactions that occur between them are shown in figure 2.2 and described below:

Main Application The main application provides the GUI that users can use for managing their logins, notes and credit cards. The GUI also allows users to configure the security of the application, manage their synced devices and browser extensions, generate new passwords, and access the information gathered by *Password Guardian*. The *Main Application* manages the life-cycle of the *Helper Application* and communicates with it via XPC [2]. The functionality of the *Main Application* is highly dependent on the services offered by the *Helper Application*.

Helper Application The helper application is the process that manages the access to the encrypted user data. It communicates with the *Main Application* via XPC, and offers an HTTP server that is used in the communication with the *Browser Extension*. The *Helper Application* is always

[2] one of the RPC mechanisms offered by *macOS* [60, 61]

available from an icon in the *System Top Bar*, from where users can perform a few tasks such as: locking and unlocking their vault and opening the *Main Application*.

Browser Extension The browser extension offers assistance to the user while using the web browser. With the *Browser Extension*, users can login to their online accounts with one click. They can also generate and save strong passwords for their accounts or save and automatically fill up their credit card details.

Avast Passwords for Mac is distributed via the *Mac App Store* as well as through the distribution channels of Avast such as the *Avast Antivirus for Mac*. Users that install the *Mac App Store* version of the application receive their updates through the update mechanism enabled by Apple. In contrast, users that install the application from a different source, receive their updates through a custom updater deployed along side the application during the first install.

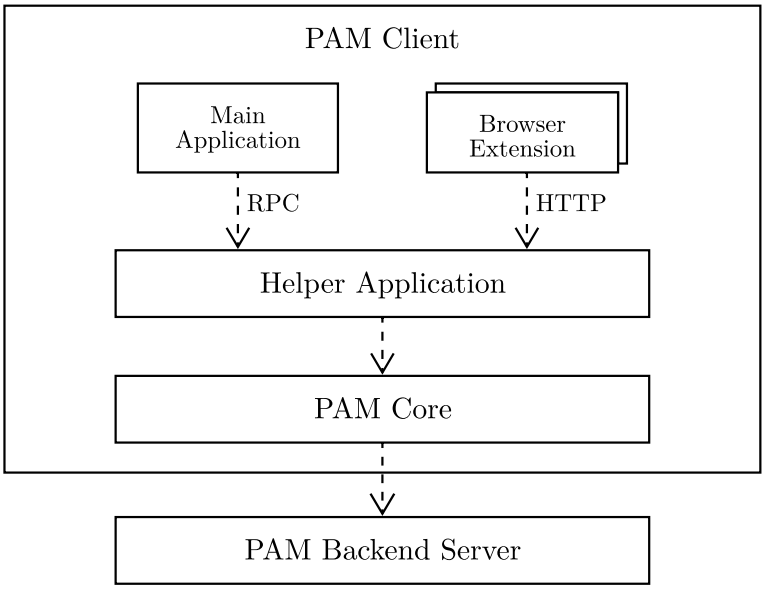

Figure 2.2: The different components of *Avast Passwords for Mac* as well as the main interactions that occur between them

2.2 Automated Acceptance Test Suite

2.2.1 Requirements

To guarantee the quality of *Avast Passwords for Mac*, different types of testing are used. Unit tests run with each build of the application and are part of the *Continuous Integration* pipeline. Exploratory tests are performed by QA engineers most often during the development of new features to check the usability of the application and to look for improvements. Acceptance tests are conducted for each release candidate and also during the development of new functionalities to confirm that all functional requirements are met.

The most time consuming QA process is acceptance testing. This process, currently requires manually verifying several different scenarios which lack proper definition and in most cases are not documented. Testing all the scenarios can take QA engineers several hours.

This process, however, can be made more efficient by implementing an automated acceptance test suite.

Before diving into the of the automation process, is important to define the high level requirements of the test suite.

In general, a good acceptance test suite should meet three general requirements:

Running against the GUI Acceptance tests should simulate the behavior of the end user. To ensure the behavior is as accurate as possible, the tests should interact with the application in the same way an end user would. In the case of an application with GUI, the acceptance tests should be run directly against it. If the tests run directly against the business logic, some errors in the UI or in the interactions between the business logic and the GUI would remain undiscovered. [2, 10]

Clean Environment Acceptance tests should be executed in a production like environment. This means that the application should be tested on a machine whose configuration and environment are as similar as possible to the ones end users will run it on. This is necessary because, frequently, the tools used by developers pollute the testing environment to the extent where successfully running the application on a developer's machine gives no guarantees as to whether the application will work correctly in production. For this purpose, virtual machines with clean operating systems are generally used. [2]

Binary Integrity Every time the code is compiled, there is a probability of introducing differences in the final binary. For example, the code may have been changed unintentionally, some settings may have changed in the build machine, or a different compiler version may have been used. Therefore the application binary used during acceptance testing should

be the same binary that will be eventually be promoted to the production environment. This, so-called *binary integrity*, guarantees that the binary released to production is the exact same binary that was tested, and thus reduce the probability of delivering untested software. [2]

In summary, a good acceptance test suite: strives to behave like a regular user by interacting with the application via the same channels an end user would; is executed on a clean machine that resembles the production environment; uses the exact same binary that will be eventually promoted to production.

Besides these general requirements, five additional requirements have been specified for the automated acceptance test suite for *Avast Passwords for Mac*:

Readability and maintainability Writing new tests should be easy and comparatively faster than manually testing the same scenarios. Moreover, changes in the code of the application do not change any functional requirements should not affect the results of the tests. Similarly, changes in the specification of the requirements should be easy to implement in the test suite.

Reliability The result of the tests should not contain any false positives. Passing all the tests should mean that the application does not contain any important errors and is ready to be released.

Language Independence *Avast Passwords for Mac* will be available in 19 languages in the near future. The automated acceptance test suite should be able to verify the functionality of the application in all these languages. For this reason, tests must not be dependent on the text in the labels of the UI.

***macOS* Compatibility** The automated acceptance test suite must be able to run on all the operating system versions supported by *Avast Passwords for Mac*, namely: *OSX El Capitan*, *macOS Sierra* and *macOS High Sierra*.

Tests Shareability Because most of the functional requirements are shared across platforms, it should be possible to share at least a fraction of the test definitions. Consequently, when a new feature is developed, the definitions of the tests could be written only once for all the platforms while keeping the implementation details different and specific for each platform.

2.2.2 Architecture

As discussed in section 1.2.3, a layered architecture may be very helpful in keeping the automated acceptance test suite maintainable. The layers should

be loosely coupled allowing different technologies and even languages to be used in each of them. This approach has been taken for developing the automated acceptance test suite for *Avast Passwords for Mac* application. The figure 2.3 shows the layering of the test suite along with the tools selected for each of them.

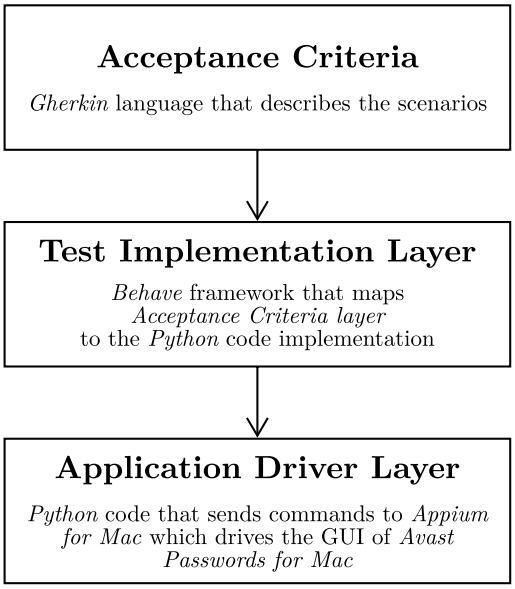

Figure 2.3: The three layers of maintainable automated acceptance tests along with the tools that can be used, adopted from [2, p. 191 (modified)]

2.3 Tools Selection

2.3.1 Acceptance Criteria

The *Gherkin* language will be used for writing the *acceptance criteria*. First, writing the *acceptance criteria* in an English-like language fulfills the requirement of *Readability and Maintainability* of the test suite. Non-technical and business stakeholders of the project can easily contribute and update the *acceptance criteria*. Second, given that *Avast Passwords for Windows* has already implemented an automated acceptance test suite with the use of *Gherkin* language, this fulfills the *Tests Shareability* requirement. With *Gherkin*, it would be possible to share the test definitions across the platforms supported by *Avast Passwords* to the extent where the `.feature` files could be written only once and the platform specific details could be managed by the *Application Driver layer*.

2.3.2 Test Implementation Layer

Python is the most widely used programming language among QA engineers in Avast. Because of this, and due to its ease of use *Python* is an ideal choice for the *Test Implementation Layer*; not to mention the extensive functionality that has already been implemented by other QA teams using the language.

Besides preserving the company preference of programming language for QA tasks, *Python* is also an appealing choice because there are several tools for mapping *Gherkin* `.feature` files into *Python* code. The tool chosen for *Avast Passwords for Mac* is Behave [26]. Not only is it easy to install using the *Python* package manager *pip*, but also is easy to use from the command line and has a good documentation [26]. Furthermore, it has been successfully used for the acceptance test suite of *Avast Passwords for Windows*.

Finally, IDEs such as *PyCharm* have native *BDD* support and offer auto-completion for writing the *acceptance criteria* as well as support for linking the criteria to the implementation code [62].

2.3.3 Application Driver Layer

An important part of the *Application Driver Layer* appertains to the automation of the GUI. Four technologies have been tested for driving the GUI on *macOS – Xcode UI Testing*, *AppleScript*, *JavaScript for Automation* and *Appium for Mac*. These tools have been evaluated and compared based on the requirements of the acceptance tests suite described in section 2.2.1. What follows is a recount of the experiences had with each of the tools, highlighting their positive and negative aspects.

Xcode UI Testing

Xcode UI Testing is the most recent addition to the Apple test toolkit. It allows developers to write their tests using languages that are familiar to the developers of *Avast Passwords for Mac* like *Swift* or *Objective-C* [37]. This makes it a strong candidate from the perspective of *Readability and Maintainability*.

Similarly, by using the accessibility identifiers in the UI widgets of *Avast Passwords for Mac*, the tests written with this tool can be decoupled from the text of the the labels in the application thus confirming the appeal of *Xcode UI Testing* from the perspective of *Language Independence* [37].

Up to this point, *Xcode UI Testing* appears as the go-to solution for implementing the *Application Driver Layer* of the acceptance test suite. After a more careful inspection, however, four particular issues make the *Xcode UI Testing* unusable for the automated acceptance tests of *Avast Passwords for Mac*.

First, just like unit tests, *Xcode UI tests* can only be run from within *Xcode* [63]. This runs against the *Clean Environment* requirement, because

it introduces noticeable environment changes that will not be available in the actual production environment.

Second, the application binary needs to be built along with the tests in order to execute them [63]. This setup goes against the *Binary Integrity* requirement, because it requires a binary that is created for the purpose of acceptance testing instead of running against the release candidate binary that will later be promoted to production,

Third, because *Xcode* needs to be able to compile the application in order to run the tests, some compatibility issues appear when trying to run the tests on older operating systems. For instance, *OSX El Capitan* can not run the latest version of *Xcode* [64], and is thus unable to compile and run the tests over *Avast Passwords for Mac*.

Forth, based on experience with using *Xcode UI Testing*, the tests are not so reliable, reporting a considerable number of false positive test results. On the other hand, it is possible to easily debug the tests in *Xcode*.

In summary, *Xcode UI Testing* is not suitable for automated acceptance tests for *Avast Passwords for Mac* project because: the tests can only be run on a machine with a compatible *Xcode* installation; they can not be executed against the production binary; they can not run on older versions of *macOS*; they report too many false positives.

AppleScript

Although *AppleScript* is a rather old technology (included in *macOS* since 1991 [65], it offers interesting possibilities.

First of all, due to its age, it is virtually ubiquitous on *macOS* systems. This means that no additional software is required for running the acceptance tests; very much in keeping with the *Clean Environment* requirement. Similarly, it means that *AppleScript* does not suffer from the compatibility issues suffered by *Xcode UI Testing*, standing as a strong candidate from the perspective of the *macOS compatibility* requirement.

Next, because *AppleScript* is part of *macOS* system, it is able to drive the GUI of any executable application without running against important security policies. This allows the tests to be executed against the binary intended for production fulfilling the requirement of *Binary Integrity*.

AppleScript, however, has several problems.

The first problem is the programming language and its syntax. *AppleScript* is quite different from other programming languages. It uses an English-like syntax that allows non-programmers to use the language [47]. Paradoxically, the same syntax makes it hard to understand and maintain for software engineers, who are familiar with Object Oriented Languages like *Swift*. Additionally, the language imposes important constraints on the programmers, for example, only strings and numbers can be passed as parameters to a function,

not objects. For this reason, *AppleScript* does not fulfill the *Readability and Maintainability* requirement.

The second issue is that unless the *application under test* is modified to be able to react to specific *Apple Events*, the possibilities offered by *AppleScript* for driving the UI of the application are limited. One alternative to get around these limitations, is to make the *application under test* scriptable and responsive to *Apple Events* [46]. This, however, implies that for every UI widget (e.g: buttons, text fields, labels, etc.) the tests need to interact with, there should be a matching *Apple Event*. This not only introduces considerable complexity to the production code only for the purpose of acceptance testing, but also means that the acceptance tests would not run against the actual GUI of the application, but rather against a different layer that achieves similar effects as the interactions with GUI.

Another alternative to get around the limitations of *AppleScript*, without requiring important modifications to the source code of the *application under test* is to drive the GUI of the application using an intermediate process called *System Events* [45]. Using this process, an *AppleScript* can control the GUI of any application by leveraging the accessibility features of *macOS*. However, the script can only access the UI elements based on the accessibility tree structure and sometimes also the labels. This is a deficient approach, because it makes the tests highly dependent on the UI structure and the localization of the strings in the application; if any of these changed even slightly, the tests would break. This is a clear infringement over the *Language Independence* requirement.

Here is an example of clicking the *Add* button using the *System Events* process. In this case it is impossible to locate the button based on the label, because it does not have any. Instead, it has a *static text* element inside.

```
tell application "System Events"
  tell process "Avast Passwords"
    click button 3 of window 1
  end tell
end tell
```

Writing and specially debugging an *AppleScript* is rather cumbersome. There is limited official documentation and since the practitioners community is rather small there are very few unofficial sources. Finally, the project has not received any major updates since *OSX El Capitan* and seems to rank very low in the development priorities of Apple [65].

JavaScript for Automation

Because *JavaScript for Automation* uses the same core technology as *AppleScript* [44], it shares all of its positive aspects. These include: keeping the testing environment clean, preserving binary integrity and having no compatibility issues. Furthermore, and unlike *AppleScript*, the *JavaScript* programming language is much more familiar to QA engineers and developers, making it a better choice from the perspective of *Readability and Maintainability*.

Despite its benefits, *JavaScript for Automation* has issues of its own.

First of all, and as it was the case with *AppleScript*, UI elements are located using the accessibility tree and the text of the labels of the application. This goes against the requirement of *Language Independence*. To get a glimpse of the issue, the next snippet shows the JavaScript function used for uninstalling *Avast Passwords for Mac* using the GUI. The snippet highlights the tight coupling that appears between the structure of the GUI and the labels in it with the code of the tests. The example also shows the implementation of the `waitAndClickButton` function which waits until the specified button appears in the GUI and then clicks on it. This is necessary because *JavaScript for Automation* does not provide any built-in API for waiting for the UI elements to appear, which is a very common requirement when implementing an automated acceptance test suite.

```
function waitAndClickButton(button,seconds) {
  var startTime = Application.currentApplication().currentDate()
  while ( ! button.exists() || ! button.enabled())
    if ((app.currentDate() - startTime) > (seconds * 1000))
      throw new Error("button was not found")
    delay(0.2)

  button.click()
}

function clickUninstallButton() {
  // Activate the window
  Application('Avast Passwords').activate()

  // Click on check for updates in Avast Passwords menu
  var proc = Application('System
  ↪  Events').processes.byName('Avast Passwords')
  var helpMenu = proc.menuBars[0].menuBarItems.byName('Help');
  var uninstallMenuItem =
  ↪  helpMenu.menus[0].menuItems.byName('Uninstall...');
  waitAndClickButton(uninstallMenuItem, 5)
```

```
  // Checks if window "Uninstall" appears
  var uninstallWindow = proc.windows.byName("")
  if ( ! wait(uninstallWindow, 5) ||
  ↪  uninstallWindow.staticTexts[0].value() != "Uninstall" )
    throw new Error("Window Uninstall was not found!")
}
```

The second issue regards the available tooling and documentation. Although the process and tools available for writing tests in *JavaScript for Automation* are very similar to ones available for *AppleScript*, when using JavaScript, developers can debug their scripts using *Safari* [48]. This seems like a good thing. Unfortunately the debugger is very unstable and crashes frequently. As for the documentation, the official resources are rather shallow and introductory and the unofficial documentation is very scarce.

Appium for Mac

Because *Appium for Mac* is an HTTP server listening on a specific port and interacting with he application under test by leveraging the *Accessibility API* [52], it does not modify the testing environment as much as *Xcode UI Tests* would.

Because it leverages the *Accessibility API* for controlling the GUI of the application under test, *Appium for Mac* can interact with the production binary and does not require a special build for the purpose of acceptance testing.

The client–server architecture of *Appium for Mac* allows QA engineers to use various languages for writing the tests, including *Python* for the *Application Driver Layer*. This makes the tests easy to write and maintain.

Appium for Mac runs on all *macOS* versions supported by *Avast Passwords for Mac* and can therefore be used for testing on all of them. In addition, a single machine can orchestrate the execution of the tests on several different machines with different versions of *macOS*.

While the *Selenium WebDriver API* supports several methods for locating UI elements, queries by ID and XPath are the only ones implemented in the *Appium for Mac* driver. An XPath expression represents a path through the accessibility tree of an application all the way down to a specific UI element. These XPaths can be deduced using developer tools such as the *Accessibility Inspector* which is bundled with *Xcode*. Alternatively, *Appium for Mac* offers a feature for getting the XPath of a UI element by simply clicking on it. For example, the XPath provided by *Appium for Mac* for the *Add Login* button looks as follows:

```
add_button_xpath = "/AXApplication[@AXTitle='Avast
↪  Passwords']/AXWindow[@AXTitle='Avast Passwords' and
↪  @AXIdentifier='main_window' and
↪  @AXSubrole='AXStandardWindow']/
↪  AXButton[@AXIdentifier='pl_top_add']"
```

Even though they are very effective, several issues appear when relying on XPath expressions for finding elements in the GUI of the the application under test.

First of all, because they represent the path through the whole accessibility tree of the application, XPaths may be hard to read and maintain. This problem can be somewhat alleviated by disconnecting parts of the XPath and concatenate them using basic string operations. For example:

```
application_xpath = "/AXApplication[@AXTitle='Avast Passwords']"

main_window_xpath = application_xpath +
↪  "/AXWindow[@AXTitle='Avast Passwords' and
↪  @AXIdentifier='main_window' and
↪  @AXSubrole='AXStandardWindow']"

add_button_xpath = main_window_xpath +
↪  "/AXButton[@AXIdentifier='pl_top_add']"
```

A second issue emerges because, by definition, XPaths are tightly coupled with the structure of the accessibility tree of the application, which means that if the UI structure changes, even slightly, all XPaths used in the tests would need to be modified.

The second option for locating UI elements when using *Appium for Mac* is to use queries by IDs. These IDs are accessibility identifiers that need to be set in the code of the application. Tools like *Accessibility Inspector* can help to reveal the accessibility identifiers of the UI elements. Figure 2.4 shows the *Accessibility Inspector* in action while locating a UI element. Using the provided information is possible to determine the accessibility identifiers and also the tree structure (both shown in red boxes).

Three aspects should be considered when using this approach. First, when locating a UI element using the ID, the whole UI tree needs to be searched, which makes this approach a bit slower than locating elements using their XPath. However in case of an application with a relatively simple UI structure, the time complexity is negligible. Second, the accessibility identifiers need to

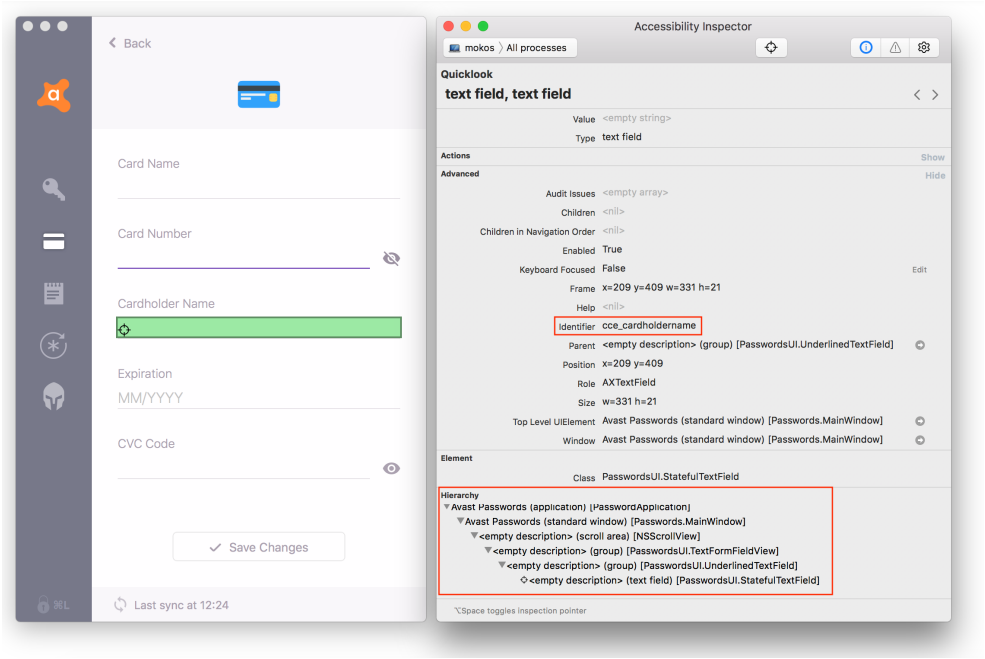

Figure 2.4: Usage of the *Accessibility Inspector* tool to determine the accessibility identifiers of the UI elements

be set for each UI Element directly in the source code of the application. Third, the accessibility identifiers need to be unique in order to reliably find the right UI element.

Finally, here are some closing remarks regarding the overall *Appium for Mac* project as well as the tooling and documentation available.

Appium for Mac is a relatively active open source project with regular contributions. Because the project is open source, it is also relatively easy to implement modifications with the purpose of fixing bugs or adding new functionality; this was, in fact, what was done to implement a couple of test scenarios of *Avast Passwords for Mac*[3].

As for the process and tooling available for writing tests using *Appium for Mac*, the landscape appears more favorable. Tests can be written using *Python* and mature IDEs like *PyCharm* [66] can be used for writing and debugging the code.

Finally, because *Appium for Mac* is built on top of seasoned projects like the *Selenium WebDriver API*, and *JSON Wire Protocol* the available documentation is deep and extensive.

[3]The contributing to the *Appium for Mac* project is discussed in section 3.3

Criteria	GUI Driving Tools			
	Xcode UI Testing	AppleScript	JavaScript for Automation	Appium for Mac
Running against the GUI	✓	✓	✓	✓
Clean Environment	✗	✓	✓	✓
Binary Integrity	✗	✓	✓	✓
Ease of writing and maintaining	✓	✗	✓	✓
Experienced Reliability	✗	✓	✓	✓
Language Independence	✓	✗	✗	✓
macOS Compatibility	✗	✓	✓	✓

Table 2.1: Selection of the tool for the application driver layer of *Avast Passwords for Mac*

Tool Selection

Each of the tools described in the previous section, has been tested to find out the compliance with the acceptance tests requirements defined in section 2.2.1. The table 2.1 shows the list of requirements along with the evaluated tools. The table shows a *check mark* when the tool meets the requirement or an *X mark* when it doesn't. As shown in the table *Appium for Mac* is the tool that seems to fit more tightly the requirements of the acceptance tests of *Avast Passwords for Mac*.

Chapter 3

Design & Realization

This chapter describes the implementation of the automated acceptance test suite for *Avast Passwords for Mac* using the tools selected in chapter 2.

The tools selected for implementing the automated acceptance test suite are:

- *Gherkin* language used for writing the *acceptance criteria*
- *behave Python* framework used for linking the criteria to the application driver layer
- *Selenium WebDriver API* for implementing the application driver layer and communicating with the GUI driver
- *Appium for Mac* for driving the GUI of the application on *macOS*

3.1 Test Suite Implementation

3.1.1 Acceptance Criteria

The *acceptance criteria* were specified using the *Gherkin* language. They were written keeping in mind two goals: automation and lose coupling with the implementation details of the tests. For example, they do not contain steps like `I click the Add button` or `I click on the Create Vault button`. Instead they describe the steps using a higher level of abstraction. For instance, `I add a new login` and `I have a vault`.

To give a wider example, consider a feature from *Avast Passwords for Mac* like retrieving and creating a new login. The *acceptance criteria* (taken from the *Gherkin* `logins.feature` file) looks as follows:

```
Feature: Logins
  # Tests the storing and retrieval of the logins

  Scenario: User should be able to create and retrieve the login
    Given I have vault
    And I am on logins screen
    When I add new login
    And I fill login service name as Test service
    And I fill login url as example.com
    And I fill login username as testuser
    And I fill login password as passwd
    And I fill login note as Login note
    And I save the login
    Then login is saved
    When I open details of login Test service
    Then service name in login detail is Test service
    And url in login detail is example.com
    And username in login detail is testuser
    And password in login detail is passwd
    And note in login detail is Login note
```

The *Gherkin* language offers a feature called *Scenario Outlines*. This feature reduces code repetition by allowing the definition of a table with examples, each of which can be referenced via placeholders in the scenarios. In the following example, an *Scenario Outline* is used for verifying that the correct credit card type is displayed for a given credit card number:

```
Feature: Credit cards
  # Tests the storing and retrieval of the credit card details

  Scenario Outline: The types of cards are displayed correctly
    Given I have vault
    And I am on credit cards screen
    When I add new credit card with card number <card_number>
    Then credit card is saved
    And Credit card <name> exists

    Examples:
      | card_number       | name                |
      | 4539049282009264  | Visa •••9264        |
      | 5247033672052399  | MasterCard •••2399  |
```

```
|  6011352096160097  |  Discover •••0097    |
|  340267047080100   |  Amex •••0100        |
|  6759649826438453  |  Maestro •••8453     |
|  36700102000000    |  Diners Club •••0000 |
|  3528000700000000  |  JCB •••0000         |
```

3.1.2 Test Implementation Layer

The implementation of the steps defined in the `.feature` files lays in corresponding *Python* files that the *behave* framework links with the `.feature` files. These steps definitions need to be located in the same directory as the `.feature` files, which can be seen in figure 3.1. The majority of the steps from the previous example are backed up by the `logins.py` file.

The steps definitions call directly the application driver layer through the `ap = AvastPasswordsProxy()` object that will be discussed in section 3.1.3. Some example steps definitions are shown below:

```
from behave import given, when, then
ap = AvastPasswordsProxy()

@given("I am on logins screen")
def step_impl(context):
  ap.main_screen().navigate_to_logins_screen()

@when("I open details of login {service_name}")
def step_impl(context, service_name):
  ap.main_screen().logins_screen().logins_list_screen().
  ↪  select_login_by_service_name(service_name)

@then("username in login detail is {username}")
def step_impl(context, username):
  assert_that(ap.main_screen().logins_screen().
  ↪  logins_detail_screen().get_username(), equal_to(username))
```

The `environment.py` module defines actions that happen before and after certain events. For example, it defines actions that happen before and after running the whole test suite as well as actions that should take place before and after specific `.feature` files and specific scenarios. In this case, the `environment.py` contains the following functions that are executed by the *behave* framework automatically:

```
def before_all(context):
  context.config.setup_logging()
  driver.get()
  ap.open_app()
  ap.new_vault()

def after_all(context):
  driver.close()

def before_scenario(context, scenario):
  ap.open_app()

def after_scenario(context, scenario):
  ap.close_app()
```

The `before_all` method is executed before the whole test suite and takes care of setting up the logging, creating the *WebDriver* instance and getting the application to a completely clean state. The `after_all` method is executed at the end of the test suite and closes the *WebDriver* instance. The methods `before_scenario` and `after_scenario` make sure that the application is restarted between scenarios.

The whole folder structure of the the *Gherkin* `.feature` files and corresponding steps implementations is shown in figure 3.1.

```
features ..... The folder with the .feature files
  logins.feature
  credit_cards.feature
  notes.feature
  generator.feature
  environment.py
  steps ..... The folder with the steps implementation
    general.py
    logins.py
    credit_cards.py
    notes.py
    generator.py
ap_driver ..... The folder with the application driver layer
```

Figure 3.1: Folder structure of the *Gherkin* `.feature` files and corresponding steps implementation

3.1.3 Application Driver Layer

The figure 1.2 shows the layering of the acceptance test suite. The implementation of the test suite follows this approach and adds a little more complexity. Figure 3.2 describes in more detail the architecture of the actual implementation of the automated acceptance test suite for *Avast Passwords for Mac*.

Appium for Mac is, at its core, an HTTP server. For communicating with it, the test suite needs to instantiate a *Selenium WebDriver* object, which then acts as proxy for all the communications with the *Appium for Mac* server. Because this *WebDriver* instance is needed in many places in the *application driver layer*, it is offered as a singleton object through the wrapper `driver.py`. The IP address of the runner machine is set in a configuration file `config.json`

Interacting with the UI Elements

To locate the UI elements in the application, the test suite relies on the accessibility identifiers put in place in the application source code. For this purpose, the `Accessibility.swift` file was added to the source code of *Avast Passwords for Mac*. This file contains all the accessibility identifiers used in the application organized in nested *enumerations*. Using the Apple API `NSAccessibility.setAccessibilityIdentifier(_: String?)`, the identifiers were set for all UI elements that the tests need to interact with.

The *WebDriver API* supported by *Appium for Mac* offers several methods for locating and interacting with the UI elements of the application under test. In the *Python* implementation of this API, the methods used for locating the UI elements are `WebDriver.find_element_by_id(id)` and `WebDriver.find_element_by_xpath(xpath)`. These methods return a *WebElement* object which has methods like `click()` or `send_keys(keys)`.

Appium for Mac offers two options for clicking on the UI elements. The first one is implemented via the accessibility features of *macOS* and clicks the element without moving the mouse cursor to it. This is achieved by the `click()` method of the *WebElement* instance. The second option moves the mouse cursor to the UI element first, which simulates more closely the interaction of a real user. To achieve this behavior, the `ActionChains` class needs to be used. An example of clicking the UI element with this approach could be this: `ActionChains(WebDriver).click(WebElement).perform()`

To manage this additional complexity, proxy objects that wrap the *WebElement API* were created. These *WebElement* proxies implement all the methods needed for interacting with the UI elements. These methods are divided into specific subclasses of the `Element` class. The `Element` class implements general methods for interacting with all types of UI elements. For example, it offers methods for locating an element, clicking on it, moving the mouse cursor etc.

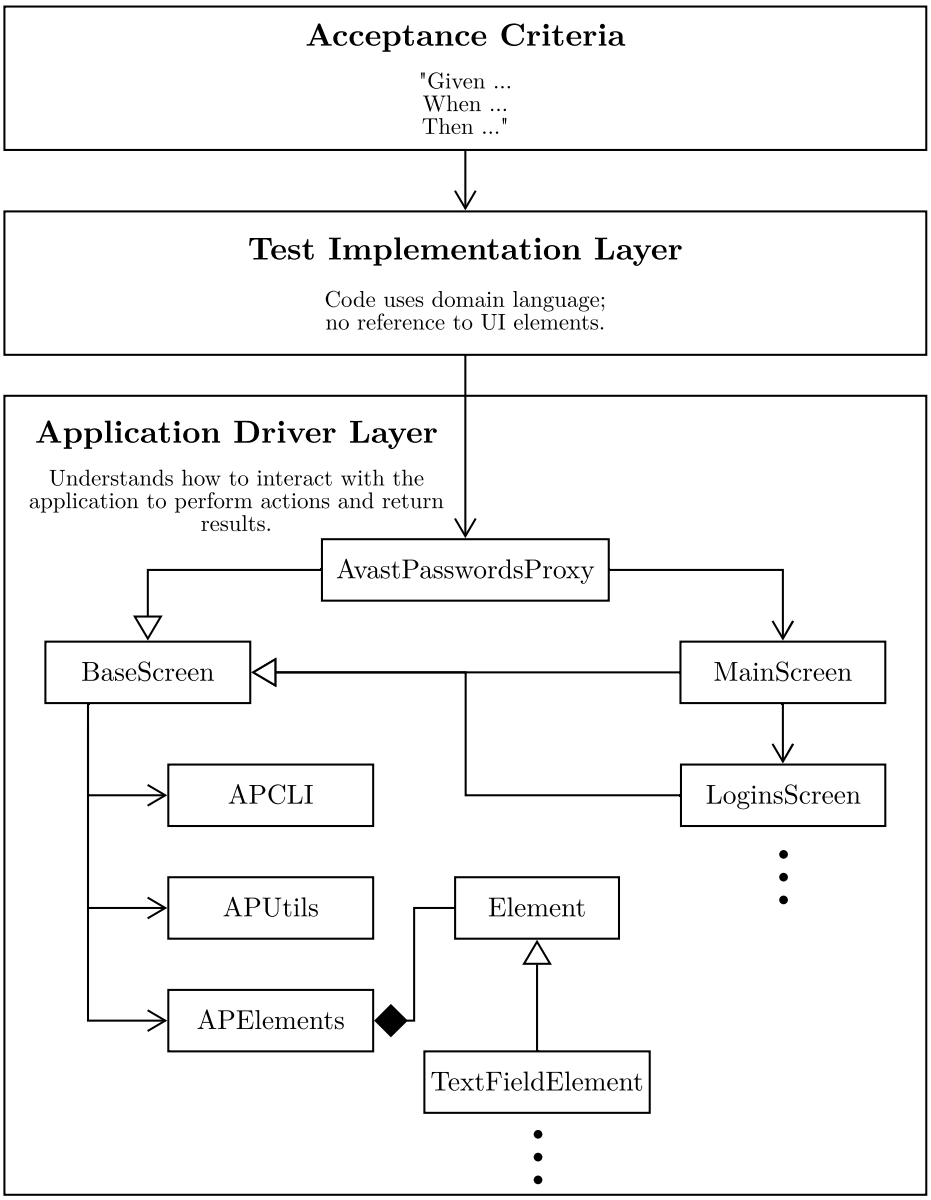

Figure 3.2: The domain model of the *Application Driver Layer*

There are also more specific *WebElement* proxies. For instance, the `CheckboxElement`, which represents a checkbox, has three additional methods. These methods offer a high level API for selecting, deselecting and querying the state of a checkbox. Another example is the `TextFieldElement`, which

offers an API for clearing a text field and setting its value.

In contrast, the `ListElement` class, acts as a different type of proxy. This class represents a list of *WebElement* objects. It overrides the method for locating the UI element with the method for locating multiple objects. It offers high level APIs for counting the elements, getting their values etc.

An example of *WebElement* proxy classes is shown in figure 3.3. This figure shows the `Element` base class with one of its subclasses–`TextFieldElement`. All subclasses of the `Element` class can be found in figure C.1

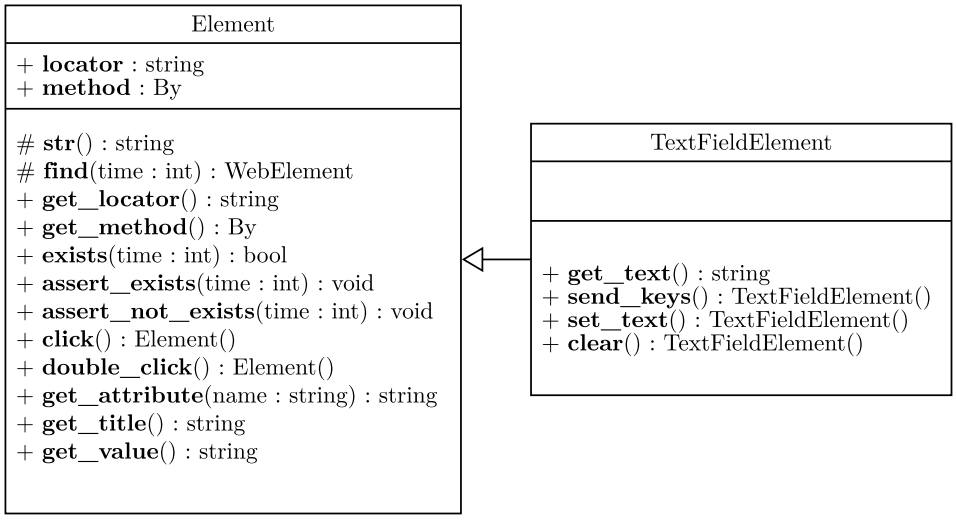

Figure 3.3: Class diagram of the `Element` class and one of its subclasses–`TextFieldElement`

Each *WebElement* proxy class is initialized with a locator and a method for locating it. The method can be either `selenium.By.ID` or `selenium.By.XPATH`. The locator is a string with either the accessibility identifier or the xpath of the element.

The protected method `_find(time)` defined in `Element` class uses the locator and the method for locating the element, waiting for it the amount of time defined in parameter and returns the *WebElement* object if the element is found or throws and exception if the element can not be located.

In `Element` class, there are also methods for checking the existence of the UI element on the screen. The method `exists(time)` returns true if the UI element can be located on the screen or false otherwise. Methods `assert_exists(time)` and `assert_not_exists(time)` throw an exception if the element could not be found or could be found, respectively.

The complete list of instances of `Element` class and its subclasses that the tests interact with is defined in the `APElements` class. The approach of nested

classes was used for better navigation in the UI elements. A few defined UI elements from the `APElements` class are shown below:

```
from selenium.webdriver.common.by import By
from ap_driver.element import *

class APElements:
  class LockedWindow:
    title = Element(By.ID, "ml_lockedtitle")
    password_field = TextFieldElement(By.ID, "ml_passwordfield")
    unlock_button = ButtonElement(By.ID, "ml_unlockbutton")

    class MainWindow:
      window = Element(By.ID, "main_window")
      logins_button = ButtonElement(By.ID, "mw_passwords")
      scroll_area = ScrollAreaElement(By.ID, "mw_scrollarea")

      class Logins:
        class List:
          title = Element(By.ID, "pl_top_title")
          add_button = ButtonElement(By.ID, "pl_top_add")
          service_names = ListElement(By.ID, "pl_servicename")

        class Detail:
          service_name_field = TextFieldElement(By.ID,
          ↪  "pe_servicenamefield")
          username = TextFieldElement(By.ID, "pe_usernamefield")
          password = TextFieldElement(By.ID, "pe_passwordfield")
```

Page Object Pattern

To decouple the test implementation layer from the implementation details of the application driver layer, the *Page Object Pattern* was used. For each screen of the application that the tests interact with, a page object class exists. These classes are named `screens`. Figure C.2 shows the complete diagram of all implemented screens.

All screen classes extend a `BaseScreen` class. The `BaseScreen` class offers three dependencies that are accessible to subclasses. These dependencies are `APElement`, `APUtils` and `APCLI` classes. The `APElement` class is described in section *Interacting with the UI Elements*. The `APUtils` class offers APIs that are used across all the screens such as `wait()`. The `APCLI` class acts as a proxy for the command line interface (CLI) of *Avast Passwords for Mac*, and

offers APIs for achieving tasks like removing the vault from the file system. The `BaseScreen` class has also the abstract method `is_shown(time)`, that all subclasses need to override. The class diagram of the `BaseScreen` class is shown in figure 3.4.

The point of interaction between the *test implementation layer* and the *application driver layer* is the `AvastPasswordsProxy` class. This class also extends the `BaseScreen` class and represents the whole application as a single screen. It offers attributes that allow easy access to the `MainScreen`, `LockedScreen` and `OnboardingStartScreen`. These attributes enable a fluent API that allows invocation chains to execute deeply nested methods. For example, the client call to access a method defined in `LoginsListScreen` would look like this:

```
AvastPasswordsProxy().main_screen().
↪  logins_screen().logins_list_screen().
↪  select_login_by_service_name(service_name)
```

Each screen class offers different methods that correspond with the functionality provided by an actual screen in the application. These methods deal with the implementation details needed to achieve a higher level tasks. For example, consider the creation of a new login. This process takes several steps, such as clicking the *Add* button, filling the required text fields and clicking on the *Save* button. All these steps are implemented in one publicly accessible method `LoginsScreen.add(...)`, accessible from the test implementation layer through the `AvastPasswordsProxy` object as follows:

```
@when("I add new login with service name {service_name}, url
↪  {url}, username {username}, password {password} and note
↪  {note}")
def step_impl(context, service_name, url, username, password,
↪  note):
  AvastPasswordsProxy().main_screen().logins_screen().
  ↪  add(service_name, url, username, password, note)
```

The class diagram in figure 3.4 shows the methods implemented in `LoginsListScreen` and `LoginsDetailScreen`. The figure 3.5 then shows the actual screens of the *Avast Passwords for Mac* application that corresponds to the *page objects* in figure 3.4.

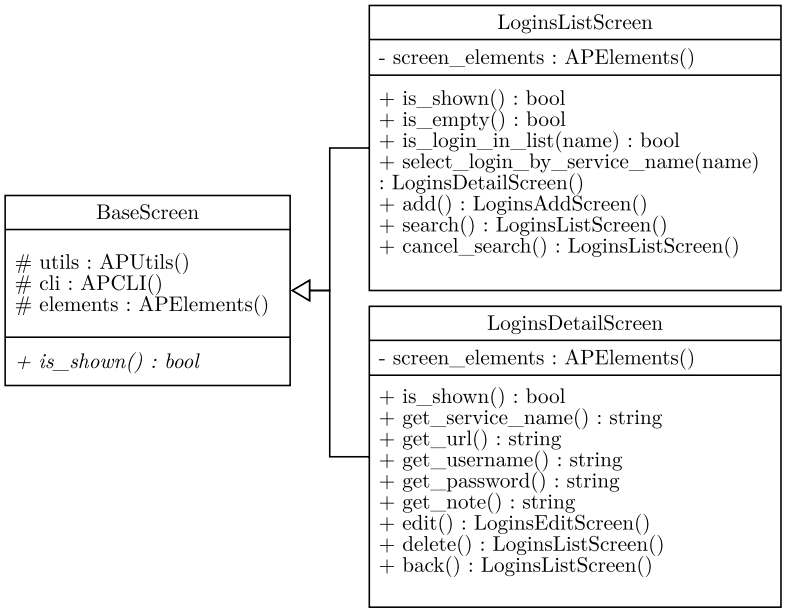

Figure 3.4: Class diagram of the *login list screen* and *login detail screen* (*page objects*) of the application

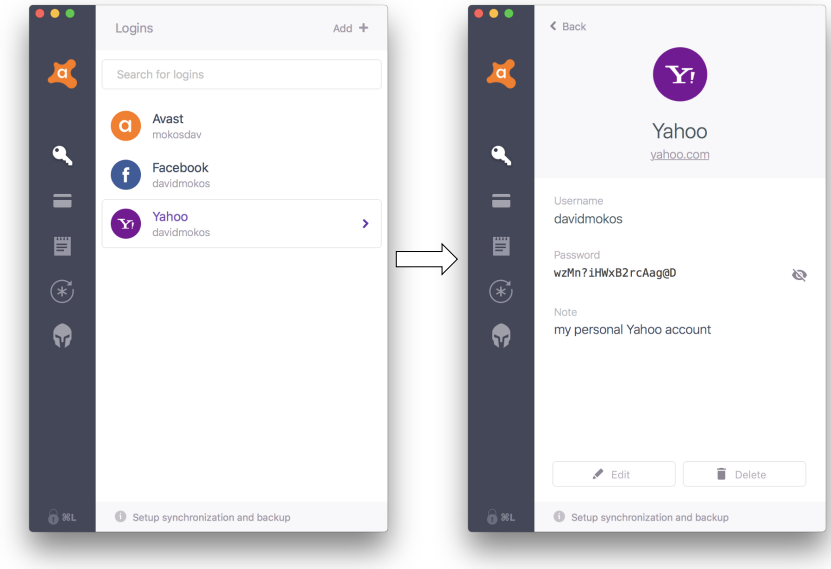

Figure 3.5: The *login list screen* and *login detail screen* of *Avast Passwords for Mac*

As stated before, the methods in the screen classes know how to interact with the application and how to achieve higher level tasks. Below, there is an example implementation of one of those methods–`get_password()` in `LoginsDetailScreen` class. This method first clicks the *Password Reveal Button* and then gets the text from appropriate text field. It uses the API provided by the *WebElement* proxy classes–`Element` and `TextFieldElement`.

```
def get_password(self):
  self.elements.MainWindow.Logins.password_reveal_button.click()
  return self.elements.MainWindow.Logins.password.get_value()
```

This layer does not implement any tests logic. It only represents an interface for achieving high level tasks that the *Avast Passwords for Mac* application offers. Therefor, in none of the screens there is an assert nor failure reporting. All asserts are part of the tests logic and are implemented in the test implementation layer.

To follow the *Page Object Pattern* completely, all methods that do not need to return any value return the class itself. To achieve this in *Python*, the `self` keyword can be used. Alternatively, if some operation results in switching of the screens, the new screen is returned. This allows clients to chain the test code like this:

```
LoginsListScreen().add().set_service_name(service_name).
↪  set_url(url).set_username(username).set_password(password).
↪  set_note(note).save()
```

3.2 Test Suite Execution

The automated acceptance test suite can be both executed and run on the machine of a QA engineer. To initiate the automatic acceptance testing, several tools need to be present on the machine:

Python 3.4 — The *Python* language in which the test code is implemented

Behave — The *Python* framework that maps the *acceptance criteria* written in *Gherkin* language to the *Python* implementation code

Selenium — *Python* language bindings for *Selenium WebDriver*

PyHamcrest The *Python* framework for better verifying the matches

Appium for Mac The application that drives the GUI of *Avast Passwords for Mac*

Avast Passwords for Mac The application under test

Also, the machine needs two important configuration changes:

Accessibility Access Allowing *Appium for Mac* the accessibility access in the system settings. This allows it to take control of the GUI of any application.

Keychain Integration Deploying the administrator password into the *macOS Keychain*. This is necessary for performing privileged operations. It is described in more detail in section 3.3.3.

After all the tools are installed and the machine is properly configured, the tests can be easily executed from the terminal using the `behave` command in the tests folder. The *Behave* framework automatically finds all the `.feature` files and executes the scenarios [26].

The results of the tests start to appear in the terminal as soon as the tests are run. The example output of the automated acceptance test suite looks as follows:

```
....

Feature: Notes # features/notes.feature:1

  Scenario: Delete note                    # features/notes.feature:36
    Given I have vault                     # features/steps/general.py:84 0.771s
    And I am on notes screen               # features/steps/general.py:34 0.882s
    When I add new note with title Note2 # features/steps/notes.py:19 3.083s
    And I open details of note Note2     # features/steps/notes.py:39 2.101s
    Then note title is Note2               # features/steps/notes.py:44 0.046s
    When I delete the note                 # features/steps/notes.py:54 1.408s
    Then note Note2 doesn't exist          # features/steps/notes.py:64 0.748s
      Assertion Failed: note Note2 exists

Failing scenarios:
  features/notes.feature:36  Delete note

3 features passed, 1 failed, 0 skipped
28 scenarios passed, 1 failed, 0 skipped
281 steps passed, 1 failed, 0 skipped, 0 undefined
Took 4m45.114s
```

3.3 Challenges

Several problems arose during the development of the automated acceptance test suite. These include problems locating the UI elements, interacting with the environment, performing privileged operations and integrating the test suite into the *Continuous Integration* Pipeline.

3.3.1 Locating UI Elements

Three blocking issues appeared while locating the UI elements:

The first one was a bug in *Appium for Mac* when trying to find more than one element using the *WebDriver API* `WebDriver.find_elements()`. This issue was resolved directly in the source code of *Appium for Mac* through a pull request[4]. This pull request has been approved and merged.

The second issue emerged when trying to locate the UI elements within a scrollable area that were out of view. To locate them, the view needed to be scrolled first. *Appium for Mac* did not implement any functionality for scrolling. This issue was solved by adding a new feature to the *Appium for Mac* source code that allows scrolling.

The third issue was caused by some elements that could not be found using the accessibility identifiers even though they could be seen using the *Accessibility Inspector*. The issue originated in a misuse of Apple *NSAccessibility API* and was solved by modifying the source code of *Avast Passwords for Mac*.

In summary, to solve problems locating the UI elements, two modifications had to be made to the *Appium for Mac* open source project and several modifications regarding the *NSAccessibility API* had to be made in the source code of *Avast Passwords for Mac*.

3.3.2 Interacting with the Environment

An important challenge appears when using two machines for executing the acceptance tests. In this setup an *executor* machine runs the python tests and transmits the GUI driving commands to a *runner* machine, which has an instance of *Appium for Mac* for receiving and executing the commands on the GUI of the application under test [5]. A significant limitation of this setup is that the tests in the *executor* machine do not have an easy way of interacting with the environment of the *runner* machine, making it very hard to achieve simple tasks such as running a *Bash* script as part of the implementation of a test.

To solve this issue, a new feature had to be added to the *Appium for Mac* project. The feature allows tests to execute *Bash* commands from the *executor* machine. These commands will be transferred through the *JSON*

[4]The pull request can be found at https://github.com/appium/appium-for-mac/pull/31

[5]This setup is further described in section 3.3.4

Wire Protocol and run on the *runner* machine via *Appium for Mac*. The pull request with the changes have been submitted[6]. While the project maintainers have not accepted the changes yet, discussions of how to improve the feature have started.

Here is the implementation of the `run_bash()` method implemented in the `APUtils` class. This method uses the implemented functionality to execute the *Bash* command in the runner machine:

```
@staticmethod
    def run_bash(bash_command):
        return driver.get().execute_script(bash_command)
```

3.3.3 Performing Privileged Operations

For some operations, such as uninstalling the application or modifying some preferences, administrator rights are necessary. In those cases, a native dialog is presented by the OS asking the user to introduce an administrator's username and password. The class `APUtils` offers a method for handling the interactions with this dialog and filling the username and password:

```
@staticmethod
def get_system_password():
  config = json.load(open("config.json"))
  return APUtils.run_bash('security find-generic-password -s ' +
  ↪  config["keyChainItem"] + ' -w')

@staticmethod
def security_agent():
  driver.get().get("coreautha")
  sec_agent = APElements.SecurityAgent
  if sec_agent.window.exists(1):
    sec_agent.password_field.click()
        sec_agent.password_field.send_keys(
        ↪  APUtils.get_system_password())
    sec_agent.ok_button.click()
  driver.get().get("Avast Passwords")
```

[6]The pull request can be found at https://github.com/appium/appium-for-mac/pull/30

Because this dialog is created by a system process called `coreautha`, the *WebDriver* instance needs to point to this process. At the end of this method, the *WebDriver* instance is pointed back to the *Avast Passwords* process.

To be able to automatically fill in the administrator password from the tests without saving these credentials in the code, the *macOS Keychain* was used [67]. The *Keychain* is a native password manager available on every installation of *macOS*. It has a CLI which allows *Python* scripts to retrieve credentials stored in it by invoking a simple *Bash* command. The retrieval of the administrator password is implemented in the method `get_system_password()`. The name of the *Keychain* item as saved in the *macOS Keychain* is stored in the configuration file `config.json`.

In summary, to perform privileged operations, the interactions with the security dialog needed to be handled. Also, to avoid storing the administrator credentials in plain text in the source of the tests, an integration with *macOS Keychain* was used.

3.3.4 Integration with the Continuous Integration Pipeline

Because *Appium for Mac* is an HTTP server, it is possible to connect to it and execute the tests from a different machine. This opens up two important possibilities. First, the *runner* machine needs only *Appium for Mac* and the application binary running, which means that the environment of machine can be kept clean. The *executor*[7] machine, on the other hand, requires *Python 3* along with *behave*, *Selenium* and *PyHamcrest* packages [8]. Second, it is possible to execute the test suite simultaneously on several machines. Consequently, this allows running the tests on all supported *macOS* versions at the same time.

While this has not been completely implemented, a proof of concept has been created. The figure 3.6 shows how the automated acceptance test suite should be executed in the *Continuous Integration* pipeline. Two bash scripts were created for this purpose. The `execute_tests.sh` copies the tests code and *Avast Passwords for Mac* binary to the *executor machine* and executes the `runner_execute.sh` script which handles the provisioning of the *runner machines*, copies the *Avast Passwords for Mac* binary there and executes the tests.

To summarize, a proof of concept for integrating the acceptance test into the *Continuous Integration* pipeline was developed and successfully tested. The proof of concept consists of two *Bash* scripts that manage the provisioning of the *runner* and *executor* machines as well as the actual execution of the tests.

[7] The *executor* machine has all the test code and executes the tests by sending commands to the *runner* machine which drives the application GUI and returns the results back to the *executor* machine

[8] These tools are described in section 3.2

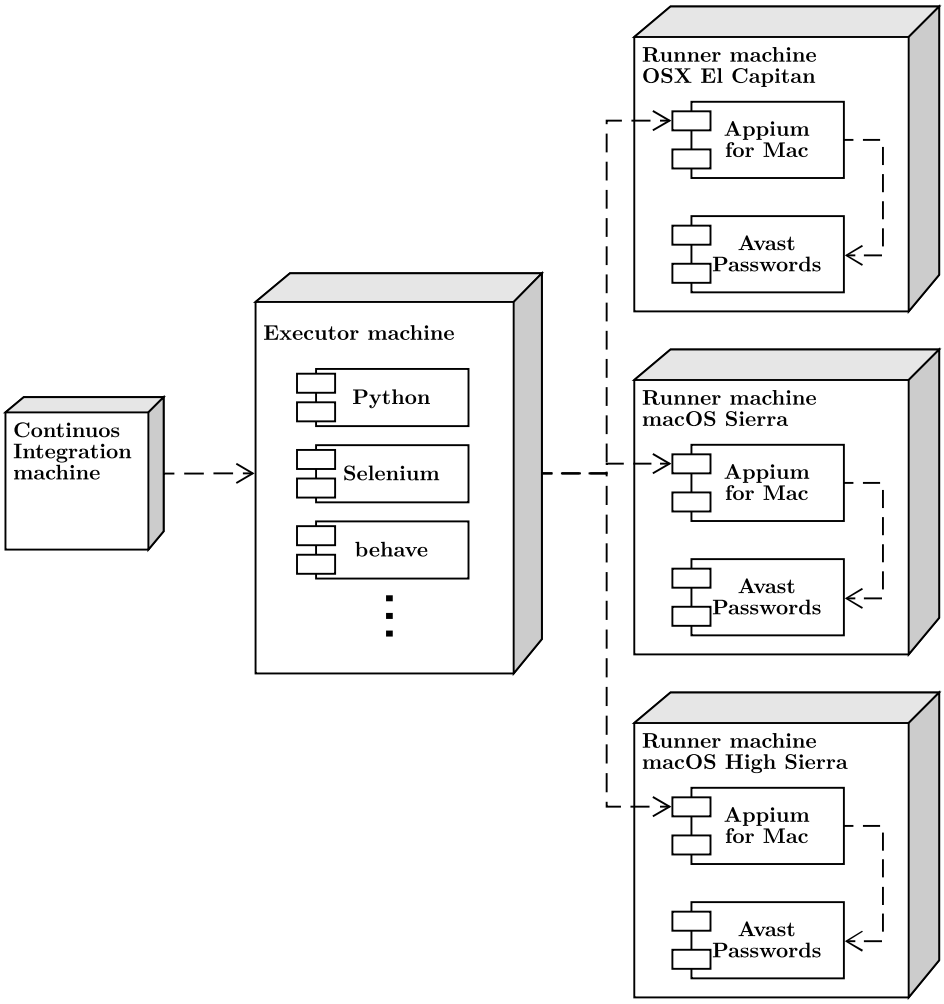

Figure 3.6: The integration of the test suite into the *Continuous Integration* pipeline

3.4 Implementation State

The current automated acceptance test suite covers approximately five eighths of the essential features of *Avast Passwords for Mac*. The tests cover:

Storing login information
Storing secure notes
Storing credit card details

The tests cover the important scenarios regarding the login information, secure notes and credit card details including storing, retrieval, editing and deletion.

Creating a new secure vault — In order to get to a clean state, the tests are able to remove the secure vault and create a new one by completing the on-boarding process, which includes creation of the new *Master Password*

Password Generator — The tests cover the generation of a secure password including setting the desired length, setting the types of characters allowed, regenerating the password and copying the password to the clipboard.

The features of *Avast Passwords for Mac* that have not yet been covered by acceptance tests are:

Synchronization across platforms — The synchronization is a complicated process that involves the interaction of several machines as well as the usage of several back-end services including a service that manages the *Avast Account* needed for the synchronization process.

Communication with Browsers — Testing of this feature would involve driving of the web browsers' GUI, including the interactions with the *Browser Extension*.

Password Guardian — Since *Password Guardian* is a premium feature, the testing of it would involve the navigation to the preferences window and managing the license subscription.

All implemented screens (*Page Objects*) of the *Avast Passwords for Mac* application can be found in figure C.2

Conclusion

The literature review of this thesis was focused on the topic of acceptance testing as part of the broader subject of software testing. In this context, acceptance tests automation was thoroughly described giving special emphasis to the potential benefits it brings to the development process as well as the common problems that arise when integrating it into the software development cycle. Later in this section, several tools for building automated acceptance tests suites on *macOS* were explored.

The practical part of this thesis focused in the implementation of an automated acceptance test suite for *Avast Passwords for Mac*. Chapter 2 evaluated the available tools for implementing acceptance test suites in *macOS*, and presented the subset of tools that were considered more appropriate for building the test suite of *Avast Passwords for Mac*. The final chapter of the thesis centered in the actual implementation of the automated acceptance tests for *Avast Passwords for Mac*, and presented snippets and examples of the tests used for covering the most important use cases of the application.

Regarding the goals of the thesis:

The main goal of implementing an automated acceptance test suite for *Avast Passwords for Mac* was achieved satisfactorily. The delivered test suite covers the most important use cases of the application and was implemented using the tools that more tightly address the requirements of the development team and the product owner.

The secondary goal of contributing to the overall testing community was also achieved satisfactorily. Three pull requests to the *Appium for Mac* open source project were submitted, each of which aimed at improving and complementing the overall functionality of the tool.

Further work

The next step in the implementation would be to complete the integration of the automated acceptance test suite into the *Continuous Integration* pipeline.

This integration would provide developers with very useful feedback while developing new features for the application.

To make the most out of the integration with the *Continuous Integration* pipeline, special attention should be given to the reporting of the test results. A relatively simple improvement, would be to add support for standard report formats like *JUnit XML*, which are also supported by the *Continuous Integration* environment used in *Avast*.

A second avenue of improvement would be the implementation of additional tests to cover the rest of the use cases of *Avast Passwords for Mac*. Features like the *Password Guardian* or the *Synchronization across platforms* may be good places to start.

Finally, an additional tool for snapshot testing should be considered. This tool would catch errors such as broken layouts and truncated texts, which the current acceptance test suite would have a hard time detecting. These types of checks will become more relevant in the near future, since there is a plan to localize *Avast Passwords for Mac* into 19 languages in upcoming versions.

Bibliography

[1] Myers, G. J.; Sandler, C.; et al. *The Art of Software Testing*. Wiley, 2011, ISBN 1118031962.

[2] Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 2011, ISBN 0321601912.

[3] Crispin, L.; Gregory, J. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009, ISBN 0321534468.

[4] Haugset, B.; Hanssen, G. K. Automated Acceptance Testing: A Literature Review and an Industrial Case Study. In *Agile 2008 Conference*, Aug 2008, pp. 27–38, doi:10.1109/Agile.2008.82.

[5] James Shore. *The Problems With Acceptance Testing [online]*. [Accessed: 2018-02-12]. Available from: `http://www.jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html`

[6] Soeken, M.; Wille, R.; et al. Assisted Behavior Driven Development Using Natural Language Processing. In *Objects, Models, Components, Patterns*, edited by C. A. Furia; S. Nanz, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-642-30561-0, pp. 269–287.

[7] Mathur, S.; Shaily, M. Advancements in the V-Model. volume 1, 02 2010.

[8] Harrold, M. J. Testing: A Roadmap. In *ICSE - Future of SE Track*, 2000.

[9] Introducing Test Automation and Test-Driven Development: An Experience Report. *Electronic Notes in Theoretical Computer Science*, volume 116, 2005: pp. 3 – 15, ISSN 1571-0661, doi:https://doi.org/10.1016/j.entcs.2004.02.090, proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004).

[10] Patton, R. *Software Testing*. Sams, 2000, ISBN 0672319837.

[11] Sardana, M.; Choudhury, T.; et al. Extensive review on software testing and pipeline testing softwares. In *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, March 2017, pp. 246–251, doi:10.1109/ICBDACI.2017.8070842.

[12] Bach, J. Exploratory Testing Explained. 2003. Available from: `http://www.satisfice.com/articles/et-article.pdf`

[13] Leung, H. K. N.; White, L. Insights into regression testing [software testing]. In *Proceedings. Conference on Software Maintenance - 1989*, Oct 1989, pp. 60–69, doi:10.1109/ICSM.1989.65194.

[14] IEEE Standard for System, Software, and Hardware Verification and Validation. *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017)*, Sept 2017: pp. 1–260, doi:10.1109/IEEESTD.2017.8055462.

[15] Wiegers, K.; Beatty, J. *Software Requirements (3rd Edition) (Developer Best Practices)*. Microsoft Press, 2013, ISBN 0735679665.

[16] Agile Alliance. *Acceptance Testing [online]*. [Accessed: 2018-03-10]. Available from: `https://www.agilealliance.org/glossary/acceptance/`

[17] Alégroth, E.; Feldt, R.; et al. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology*, volume 73, 2016: pp. 66 – 80, ISSN 0950-5849, doi:https://doi.org/10.1016/j.infsof.2016.01.012.

[18] Garousi, V.; Mäntylä, M. V. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology*, volume 76, 2016: pp. 92 – 117, ISSN 0950-5849, doi:https://doi.org/10.1016/j.infsof.2016.04.015.

[19] Bob Martin. *Automated Acceptance Testing [online]*. [Accessed: 2018-04-03]. Available from: `https://skillsmatter.com/skillscasts/4143-automated-acceptance-testing`

[20] Weiss, J.; Schill, A.; et al. Literature Review of Empirical Research Studies within the Domain of Acceptance Testing. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2016, ISSN 2376-9505, pp. 181–188, doi:10.1109/SEAA.2016.33.

[21] Ricca, F.; Torchiano, M.; et al. Using acceptance tests as a support for clarifying requirements: A series of experiments. *Information and Software Technology*, volume 51, no. 2, 2009: pp. 270 – 283, ISSN 0950-5849, doi:https://doi.org/10.1016/j.infsof.2008.01.007.

[22] IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*, July 2008: pp. 1–150, doi:10.1109/IEEESTD.2008.4578383.

[23] Solis, C.; Wang, X. A Study of the Characteristics of Behaviour Driven Development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug 2011, ISSN 1089-6503, pp. 383–387, doi:10.1109/SEAA.2011.76.

[24] Cucumber. *Cucumber Reference [online]*. [Accessed: 2018-04-07]. Available from: `https://cucumber.io/docs/reference`

[25] JBehave. *What is JBehave? [online]*. [Accessed: 2018-04-08]. Available from: `http://jbehave.org/`

[26] GitHub. *Behave [online]*. [Accessed: 2018-04-07]. Available from: `https://github.com/behave/behave`

[27] GitHub. *Cucumber – Gherkin [online]*. [Accessed: 2018-04-07]. Available from: `https://github.com/cucumber/cucumber/wiki/Gherkin`

[28] Lowell, C.; Stell-Smith, J. Successful Automation of GUI Driven Acceptance Testing. In *Extreme Programming and Agile Processes in Software Engineering*, edited by M. Marchesi; G. Succi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ISBN 978-3-540-44870-9, pp. 331–333.

[29] SmartBear Software. *TestComplete Overview [online]*. [Accessed: 2018-04-06]. Available from: `https://smartbear.com/product/testcomplete/overview/`

[30] Software Testing Help. *35+ Best GUI Testing Tools with Complete Details [online]*. [Accessed: 2018-04-16]. Available from: `http://www.softwaretestinghelp.com/best-gui-testing-tools/`

[31] GitHub - SeleniumHQ. *Page Objects [online]*. [Accessed: 2018-04-07]. Available from: `https://github.com/SeleniumHQ/selenium/wiki/PageObjects`

[32] Leotta, M.; Clerissi, D.; 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*, March 2013, pp. 108–113, doi:10.1109/ICSTW.2013.19.

[33] Apple. *What is macOS [online]*. [Accessed: 2018-05-05]. Available from: `https://www.apple.com/lae/macos/what-is/`

[34] Apple. *macOS - Security [online]*. [Accessed: 2018-05-05]. Available from: `https://www.apple.com/lae/macos/security/`

[35] Apple. *Accessibility on macOS [online]*. [Accessed: 2018-04-23]. Available from: https://developer.apple.com/accessibility/macos/

[36] Apple. *Accessibility Programming Guide for OS X [online]*. [Accessed: 2018-04-23]. Available from: https://developer.apple.com/library/content/documentation/Accessibility/Conceptual/AccessibilityMacOSX/

[37] Apple. *Testing with Xcode – User Interface Testing [online]*. [Accessed 2018-02-12]. Available from: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html

[38] Apple. *Xcode [online]*. [Accessed: 2018-05-10]. Available from: https://developer.apple.com/xcode/

[39] Apple. *Xcode – Mac App Store [online]*. [Accessed: 2018-02-13]. Available from: https://itunes.apple.com/us/app/xcode/id497799835

[40] Apple. *What's New in Testing [online]*. [Accessed: 2018-02-12]. Available from: https://developer.apple.com/videos/play/wwdc2017/409/

[41] Apple. *UI Testing in Xcode [online]*. [Accessed: 2018-02-12]. Available from: https://developer.apple.com/videos/play/wwdc2015/406/

[42] Apple. *Introduction to AppleScript Overview [online]*. [Accessed: 2018-02-13]. Available from: https://developer.apple.com/library/content/documentation/AppleScript/Conceptual/AppleScriptX/AppleScriptX.html

[43] Apple. *About AppleScript [online]*. [Accessed: 2018-02-13]. Available from: https://developer.apple.com/library/content/documentation/AppleScript/Conceptual/AppleScriptX/Concepts/ScriptingOnOSX.html

[44] Apple. *Open Scripting Architecture [online]*. [Accessed: 2018-02-14]. Available from: https://developer.apple.com/library/content/documentation/AppleScript/Conceptual/AppleScriptX/Concepts/osa.html

[45] Neuburg, M. *AppleScript: The Definitive Guide: Scripting and Automating Your Mac*. Beijing Sebastopol, CA: O'Reilly Media, Inc, 2006, ISBN 978-1449379155.

[46] Apple. *Scriptable Applications [online]*. [Accessed: 2018-02-13]. Available from: https://developer.apple.com/library/content/documentation/AppleScript/Conceptual/AppleScriptX/Concepts/scriptable_apps.html

[47] Apple. *Introduction to AppleScript Language Guide [online].* [Accessed: 2018-02-13]. Available from: https://developer.apple.com/library/content/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html

[48] Apple. *JavaScript for Automation Release Notes [online].* [Accessed: 2018-02-13]. Available from: https://developer.apple.com/library/content/releasenotes/InterapplicationCommunication/RN-JavaScriptForAutomation/Articles/Introduction.html

[49] GitHub. *JavaScript for Automation Cookbook [online].* [Accessed: 2018-02-13]. Available from: https://github.com/JXA-Cookbook/JXA-Cookbook/wiki

[50] Appium. *Introduction to Appium [online].* [Accessed: 2018-02-14]. Available from: http://appium.io/docs/en/about-appium/intro/

[51] Appium. *Getting started [online].* [Accessed: 2018-02-14]. Available from: http://appium.io/getting-started.html

[52] GitHub - Appium. *Appium For Mac [online].* [Accessed: 2018-02-14]. Available from: https://github.com/appium/appium-for-mac

[53] Appium. *The Mac Driver for OS X [online].* [Accessed: 2018-02-14]. Available from: https://appium.io/docs/en/drivers/mac/

[54] SeleniumHQ. *Selenium WebDriver [online].* [Accessed: 2018-05-05]. Available from: https://www.seleniumhq.org/docs/03_webdriver.jsp

[55] GitHub - SeleniumHQ. *JsonWireProtocol [online].* [Accessed: 2018-05-12]. Available from: https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

[56] Hans, M. *Appium Essentials.* Birmingham, UK: Packt Publishing, 2015, ISBN 978-1-78439-248-2.

[57] PFiddlesoft. *PFiddlesoft Frameworks [online].* [Accessed: 2018-05-13]. Available from: http://pfiddlesoft.com/frameworks/

[58] Avast. *Avast Passwords for Mac [online].* [Accessed: 2018-05-12]. Available from: https://www.avast.com/passwords#mac

[59] Avast. *Avast Passwords Security Model [online].* [Accessed: 2018-02-23]. Available from: http://files.avast.com/files/passwords/security-whitepaper.pdf

[60] Apple. *XPC Framework Documentation [online].* [Accessed: 2018-04-06]. Available from: https://developer.apple.com/documentation/xpc

[61] Apple. *Creating XPC Services [online].* [Accessed: 2018-04-06]. Available from: https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html

[62] Filippov, D. Feature Spotlight: Behavior-Driven Development in PyCharm. https://blog.jetbrains.com/pycharm/2014/09/feature-spotlight-behavior-driven-development-in-pycharm/, 2014, [Accessed: 2018-04-07].

[63] Apple. *Testing with Xcode – Running Tests and Viewing Results [online].* [Accessed 2018-02-12]. Available from: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/05-running_tests.html

[64] Apple. *Xcode Release Notes [online].* [Accessed: 2018-05-13]. Available from: https://developer.apple.com/library/content/releasenotes/DeveloperTools/RN-Xcode/Chapters/Introduction.html

[65] Apple. *Introduction to AppleScript Release Notes [online].* [Accessed: 2018-02-13]. Available from: https://developer.apple.com/library/content/releasenotes/AppleScript/RN-AppleScript/Introduction/Introduction.html

[66] JetBrains. *PyCharm [online].* [Accessed: 2018-04-7]. Available from: https://www.jetbrains.com/pycharm/

[67] Apple. *What is Keychain Access? [online].* [Accessed: 2018-04-16]. Available from: https://support.apple.com/guide/keychain-access/what-is-keychain-access-kyca1083/mac

Appendix A

Acronyms

API Application Programming Interface

CLI Command Line Interface

GUI Graphical User Interface

HTTP Hyper Text Transfer Protocol

PAM Password Manager

RPC Remote Procedure Call

UI User Interface

XML Extensible Markup Language

Appendix B

Contents of enclosed SD card

- `readme.txt` ..... the file with SD card contents description
- `example` ..... the folder with the acceptance test suite performance
  - `video.mp4` ..... the video of the acceptance test suite in action
  - `results.txt` ..... the output of the acceptance test suite
- `acceptance_test_suite` ..... the automated acceptance test suite
  - `sources` ..... automated acceptance test suite sources
  - `software` ..... the software necessary to execute the tests
    - `avast_passwords.pkg` ..... the *Avast Passwords for Mac* installer
    - `appium_for_mac.zip` ..... the *Appium for Mac* binary
- `thesis` ..... the directory with the thesis
  - `BP_Mokos_David_2018.pdf` ..... the thesis text in PDF format
  - `sources` ..... the directory of LaTeX source codes of the thesis

Appendix C

Diagrams

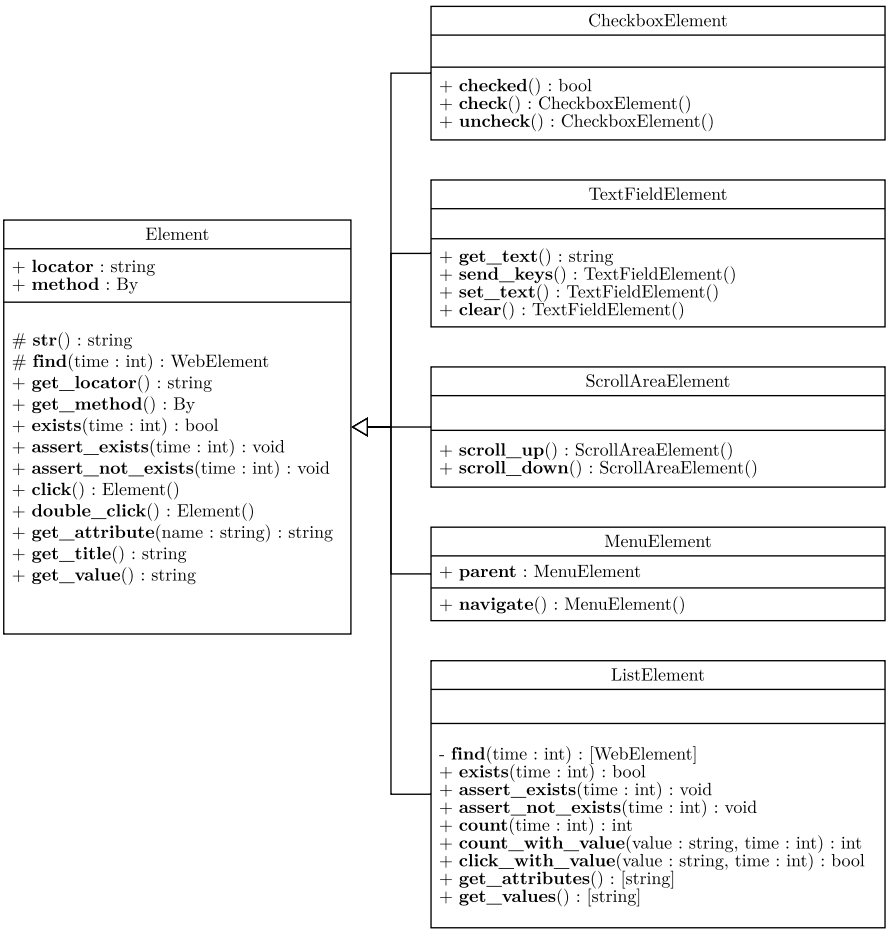

Figure C.1: Class diagram of `Element` class and all of its subclasses

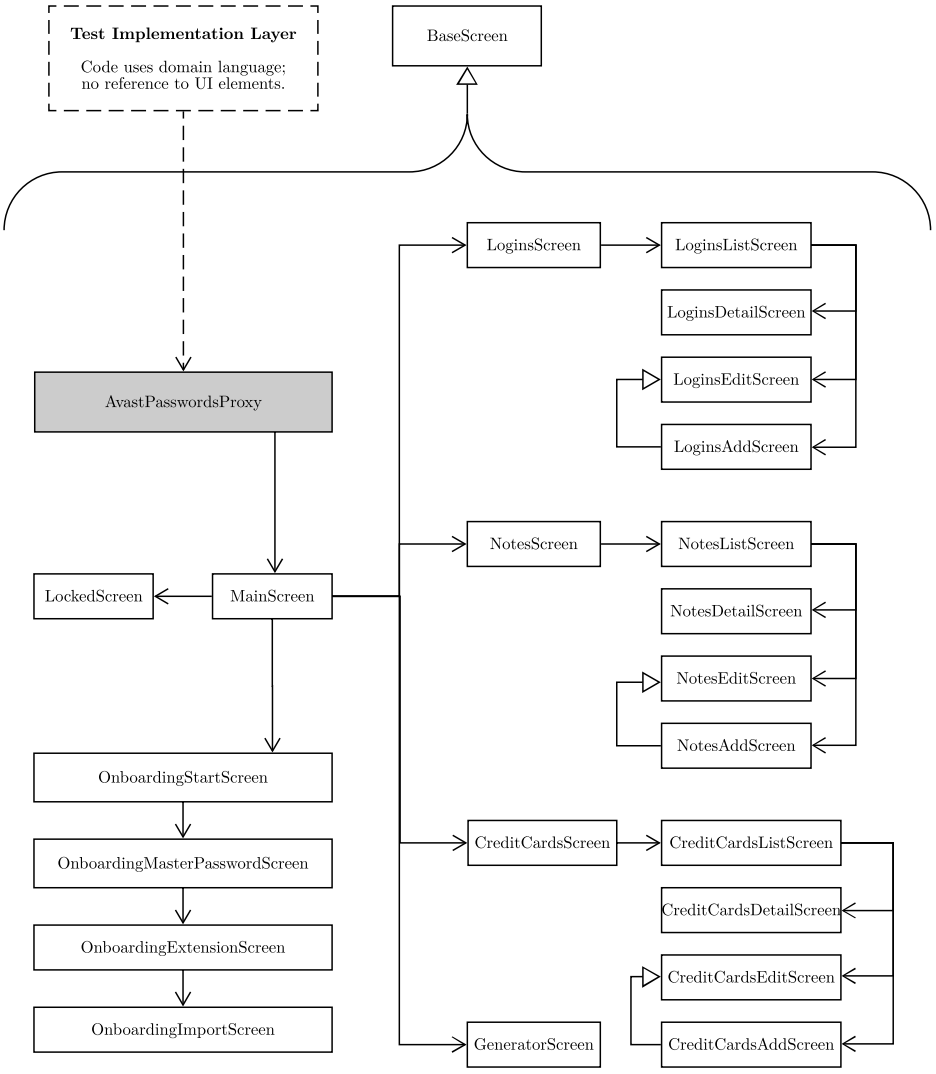

Figure C.2: All implemented screens (*Page Objects*) of the *Avast Passwords for Mac* application