Daniel Fritzsch, BSc

Integration of a Fast & Stable Automated UI Software Testing Solution into an Established iOS Application

Master's Thesis

to achieve the university degree of

Master of Science

Master's Degree Programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Institute of Software Technology

Graz, August 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Abstract

Although the automation of UI Testing for mobile applications is a trending software engineering topic today, various obstacles exist that need to be avoided. Therefore, considerations should be made in order to integrate a *fast and stable automated UI Testing* solution into complex iOS applications.

This thesis takes a fresh look at how to develop an *optimized UI Automation framework for iOS* which can be easily integrated into existing and established mobile applications. Tailored to the specific needs of the software project, the popular open-source *UI Automation framework KIF* has been enhanced and deployed for the distinctive UI Testing requirements.

The resulting automated UI Testing solution achieves a remarkably *high performance*, while providing the most *reliable testing results*. Moreover, the developed UI Automation framework is *highly maintainable and reusable* as well, because it aims at being applicable to various iOS projects. In addition to the desired high reliability and maintainability, the clear focus lies on optimizing the overall UI Testing performance for iOS apps.

First of all, this thesis provides an *introduction* to the fields of software testing and automated UI Testing for mobile iOS applications.

Afterwards, the concept of *software testing* is described in more detail, providing the foundation knowledge for the automation of UI Testing.

After the definition of software testing, the term *UI Testing* is discussed, including the automation of the corresponding UI Testing process.

Providing the required context, the *project* is summarized afterwards, describing its architecture and its main aspects related to software testing.

Following this project introduction, all the *practical achievements* are highlighted, providing an overview of the significant optimizations and extensions of the already well-established KIF UI Testing framework.

Finally, specific *implementation details* are illustrated that aim at providing a visualization of the basic concepts and optimizations.

Kurzfassung

Obwohl die Automatisierung von UI-Tests ein populäres Thema im mobilen Anwendungsbereich darstellt, existieren große Stolpersteine, die es zu vermeiden gilt. Dies ist insbesondere dann der Fall, wenn ein *schnelles und stabiles automatisiertes UI-Testkonzept* für komplexe iOS Apps benötigt wird.

Diese Arbeit behandelt die Entwicklung einer automatisierten und *optimierten UI-Testumgebung für iOS*, welche sich auf einfachste Weise in etablierte mobile Anwendungen integrieren lässt. Aufgrund der spezifischen Bedürfnisse des Projektes wurde die populäre Open Source *UI-Testumgebung namens KIF* eingesetzt und in Folge erheblich erweitert und optimiert.

Die daraus resultierende verbesserte UI-Testumgebung ist erstaunlich performant und liefert höchst zuverlässige Ergebnisse. Die gute Wartbarkeit und Wiederverwendbarkeit ermöglicht zudem den Einsatz in weiteren Software-Projekten. Der klare Fokus liegt hierbei allerdings auf einer möglichst hohen Geschwindigkeit, ohne die Stabilität und Verlässlichkeit der Testresultate zu beeinträchtigen.

Zu Beginn führt die Arbeit in die Thematik der Softwaretests ein und beinhaltet dabei auch die Automatisierung von UI-Tests für iOS Apps.

Im Anschluss wird das Konzept von Softwaretests beschrieben, um das Basiswissen für UI-Tests und deren Automatisierung zu vermitteln.

Danach wird der Begriff UI-Testen erörtert, wobei insbesondere auf den entsprechenden Prozess der UI-Automatisierung genauer eingegangen wird.

Das Projekt, inklusive der Projekt-Architektur und weiterer für das Softwaretesten relevanter Aspekte, wird im Anschluss beschrieben.

Des Weiteren werden die Erfolge der Entwicklung des UI-Testkonzeptes beleuchtet, um die signifikanten Verbesserungen und Erweiterungen der etablierten Testumgebung für iOS namens KIF aufzuzeigen.

Zu guter Letzt werden spezifische Implementierungsdetails präsentiert, um die wichtigsten Konzepte und Optimierungen zu veranschaulichen.

Contents

Abstract **v**

Figures **xiii**

Tables **xv**

Listings **xvii**

Acknowledgments **xix**

1. Introduction **1**

I. Theoretical Part **5**

2. Software Testing **7**

2.1. What Is Software Testing? 7
2.1.1. Definition of Terms 8
Informal Definition 8
Formal Definition 9
2.1.2. When Does a Bug Occur? 11
2.1.3. Why Does a Bug Occur? 11
2.2. Why Testing Software? 13
2.3. Important Aspects of Software Testing 15
2.3.1. Psychological Aspects 16
2.3.2. Economical Aspects 17

2.4. Software Development Process . . . 19
2.4.1. Development Lifecycle Models . . . 19
Big-Bang Model . . . 21
Code-and-Fix Model . . . 22
Waterfall Model . . . 23
Spiral Model . . . 25
2.4.2. Software Testing Process . . . 27
Testing with Lifecycle Models . . . 28
Who Should Test Software? . . . 30
When Testing Software? . . . 32
2.5. Software Testing Levels . . . 33
2.5.1. Black-Box Testing . . . 33
2.5.2. White-Box Testing . . . 34
2.5.3. Test Level Subdivision . . . 35
Unit Testing . . . 35
Integration Testing . . . 36
System Testing . . . 36
Acceptance Testing . . . 37
2.6. Software Testing Principles . . . 39

3. UI Testing **43**
3.1. What Is UI Testing? . . . 43
3.1.1. Definition of Terms . . . 44
3.1.2. UI Testing Categories . . . 45
3.1.3. UI Testing Layers . . . 47
3.2. Why Testing the UI? . . . 49
3.3. Important Aspects of UI Testing . . . 51
3.4. UI Testing Principles . . . 55
3.5. UI Test Automation . . . 57
3.5.1. About Automated UI Testing . . . 57
3.5.2. Automation Tools . . . 59
Tool Requirements . . . 60
3.5.3. Supporting Patterns . . . 62
Dependency Injection . . . 62
Doubles . . . 63

II. Technical Realization 65

4. Project Introduction 67
4.1. Project Architecture . . . 67
4.2. Testing Process . . . 69
4.3. Room for Improvement . . . 69
4.4. UI Testing Ambitions . . . 71

5. UI Automation in iOS 73
5.1. UI Automation Frameworks . . . 73
5.1.1. KIF - Keep It Functional . . . 74
5.1.2. UI Testing . . . 75
5.1.3. KIF VS UI Testing . . . 75
5.2. KIF Integration . . . 77
5.2.1. Testing with KIF . . . 78
5.2.2. Framework Drawbacks . . . 79
5.2.3. UI Test Optimization . . . 81
Performance & Reliability . . . 82
Understandability & Maintainability . . . 84
Additional Enhancements . . . 86
5.2.4. Remaining Issues . . . 87

6. Implementation Details 89
6.1. Text Input & Stubbing . . . 91
6.2. Scrolling & Mocking . . . 93

III. Outlook & Conclusion 95

7. Future UI Automation in iOS 97

8. Concluding Remarks 99

Bibliography 103

List of Figures

2.1. Bug Costs over Time 18
2.2. Software Development Model: Big-Bang 21
2.3. Software Development Model: Code-and-Fix 22
2.4. Software Development Model: Waterfall 23
2.5. Software Development Model: Spiral 25
2.6. Software Testing Lifecycle 27
2.7. Software Test Effort Optimum 40

6.1. Example iOS App for UI Testing 90

List of Tables

2.1. Formal Test Term Definition 10

5.1. iOS UI Automation: KIF VS "UI Testing" 76

.1. Used Acronyms Overview 109

List of Listings

1. UI Test Example: Text Input & Stubbing 91
2. UI Test Example: Scrolling & Mocking 93

Acknowledgments

As it was not always easy for me to find the motivation and time to finish my studies next to working full time, I want to thank everyone who supported me while I was writing this Master's Thesis.

Next to my family, close friends and co-workers, I want to especially highlight the contribution of my sister *Julia Fritzsch*, because she did not only read every single page, but she even created all the included images.

Moreover, special thanks go to *Sandra Hassler* for proofreading everything in a very short time, while providing very detailed and helpful feedback.

Finally, I want to thank my advisor *Wolfgang Slany* for his support and straightforward instructions which enabled me to finish in time.

Additionally, special mention goes to Keith Andrews and Karl Voit for providing the structural foundation[1] [1].

[1] LaTeX template provided by Karl Voit

1. Introduction

This thesis explains how to automate the *UI Testing process for iOS apps*, while focussing on an *outstanding performance* as well as on a very *high reliability*.

At the beginning, the scientific research of the respective field is summarized within the *theoretical part* of the thesis (Chapters 2 and 3) which is followed by the *practical realization* illustrated in Chapters 4–6. At the end, an outlook on the future of automated UI Testing in iOS as well as a few final concluding remarks are provided in Chapters 7–8.

First of all, the field of *software testing* is described in Chapter 2, including an introduction to the topic, a motivational section and the most important aspects that need consideration when testing software. Additionally, some common software development lifecycle models and the according testing process are illustrated, while explaining some of the major testing principles.

Afterwards, Chapter 3 discusses the more specific form of *UI Testing*, including an introduction to the topic, a motivational section and the most relevant aspects and principles. Moreover, the automation of UI Testing is explained where automation tools and their requirements are discussed. Additionally, supporting software development patterns are highlighted.

Chapter 4 introduces *the project* which has been used for automated UI Testing. It provides an overview of its architecture and software testing process. Furthermore, the room for improvement within this discipline is highlighted, providing the foundation for the specific UI Testing ambitions.

Following the project introduction, Chapter 5 highlights all the main achievements in the field of *automating UI Testing in iOS*. At the beginning, the major UI Automation frameworks for iOS are discussed and compared. Afterwards, the integration of the *open-source framework KIF* is explained, including a description of its standard testing procedure, its major drawbacks and potential optimizations. Especially the enhancements of this UI Automation framework are illustrated in more detail subsequently.

Chapter 6 is illustrating some *implementation details*, aiming at visualizing the optimized UI Testing approach by using the enhanced version of KIF. Therefore, two test case examples are provided, which are indicating how to properly test an iOS application in a very fast and reliable way.

At the very end of this thesis, Chapter 7 gives a *future outlook* of automated UI Testing in iOS, which discusses the currently foreseeable trends in order to make assumptions about the short-term evolution of UI Testing in iOS.

Finally, Chapter 8 is providing a rough *retrospective overview* of all the main aspects that have been described within the scope of this thesis.

Part I.

Theoretical Part

2. Software Testing

The intent of this chapter is not only to explain the basics of software testing, but also to motivate and persuade testing it properly. Additionally, it offers some insight into the most important aspects and principles as well as into the overall development and testing process of software.

2.1. What Is Software Testing?

The discipline of Software Testing has *evolved over many decades*, parallel to Software Development [2, p. 3], [3, pp. 687-690]. The *earliest article* related to testing was written *by Alan Turing in 1949* [4], which describes what is known as a so called "proof of correctness" today. The first article considered to be about software testing itself was also written by Touring in 1950 [5]. It described a special case of "How we would know that a program satisfies the requirements?", which could be seen as the foundation of software testing. Since the publication of Turing's article, *hundreds of articles and reports* about software testing have appeared [3, p. 690].

From the early 1970s on the concept of software testing started becoming more and more *associated with a professional software development process*, which is going to be explained in Section 2.4 [3, pp. 687-689]. The main reason for this *significant gain of relevance* was the steadily increasing number of computer applications. This growing *ubiquity of software* has led to rising development costs and therefore to a higher economic risk.

As even the earliest computer programs have been specified by exact input and output requirements, the logical next step was to make use of this aspect by enforcing a *"test first" software development* approach [6]. The resulting introduction of *Test-Driven Development (TDD)* by Beck in 2002 is another main reason for the renaissance of software testing in the recent past.

Although the term 'Software Testing' has been discussed and redefined several times, it is still seen as one of the *"dark arts" of software development*, according to Myers, Sandler, and Badgett [7, p. IX].

However, there still exists a generally approved *informal definition* of software testing. It is often described as the process of *testing a program with the intent of finding errors* [7, pp. 6-7]. Quality assurance, reliability estimation, validation and verification here illustrate the main purposes of testing [8, p. 213]. As there is no realistic possibility to write any program code that is completely free of bugs[1], *software always needs to be tested*[2].

Until today, "software testing has become *more difficult and easier than ever*." [7, pp. 1-2]. On the one hand there are intrinsic, well-tested routines and sophisticated test tools are available, which simplify the overall testing process described in Section 2.4.2. On the other hand, better development tools, pretested Graphical User Interfaces (GUIs), tight deadlines and a complex environment lead to an avoidance of the main testing protocols.

2.1.1. Definition of Terms

This section provides an overview of the most important terms related to software testing. Although *various names* can be associated with software testing, they only *slightly differ in meanings*, according to Patton [9, p. 14].

Informal Definition

The most *generic terms* related to software testing are *'bug', 'error' and 'problem'* [9, pp. 10-15]. The term 'bug' originated in 1947 when a big room-sized computer at Harvard called Mark II suddenly stopped working [10], because of a moth stuck between the relays. Generally, a bug represents the *differences between the required and existing conditions* of a product [8, p. 213]. The terms 'error' or 'problem' are used as synonyms for 'bug'.

[1] Term 'bug' explained in Section 2.1.1
[2] Psychological aspects described in Section 2.3.1

Also other terms as *'fault' and 'failure'* exist, which usually are related to a *more severe* or even potentially dangerous bug. What is remarkable about these two terms is that they also tend to *blame someone* to be responsible for a specific problem. In contrast to the above terms, an *'incident'* generally has a less negative meaning, rather inferring to an *unintended execution* than to a complete failure of a software product.

"Just call it what it is and go on with it" [9, p. 14]. This quote tries to tackle the *problem of overcomplicating* things through endless discussions about the correct classification of a bug, without focussing on more relevant aspects like considering how to actually handle an observed issue.

Formal Definition

Despite the fact that it is never good to think too much about how to name something appropriately, it is still necessary to provide *generally approved standards* for the most common terms and definitions.

Because of this, the *Institute of Electronics and Electrical Engineers (IEEE)* developed *two consensus international standards* on professional test engineering [3, pp. 690-691]. The first was published in 1983 [11], specifying the content of eight different test documents. Following this test documentation standard, another standard was published in 1987 [12], defining the test phases, activities, tasks and documents required for a sophisticated Unit Testing process of software[3].

In order to provide a more formal definition for the testing terms already mentioned earlier, the *International Software Testing Qualification Board (ISTQB)* published an extensive *glossary of testing terms*[4] [2, p. 3]. All terminologies shortly summarized in Table 2.1.1 are compatible with the standards established by the IEEE.

[3] Unit Testing described in Section 2.5.3

[4] http://www.istqb.org/downloads/glossary.html

Term	Definition
Error	*People* produce errors
	Synonym for *problem* or *mistake*
Bug	Error done *while coding*
Fault	*Result* or representation of an error
Failure	*Code* corresponding to a fault executes
Incident	*Symptom* associated with a failure
	Alerts user of occurrence
Test	Act of executing software with test cases
Test Case	Find failures or demonstrate correctness
	Identity associated with program behavior
	Set of inputs & expected outputs

Table 2.1.: Formal Test Term Definition

The formal definition of the most important terms associated with software testing. The terminologies are compatible with the standards of the IEEE established in 1983 [11], [2, p. 3].

Also, Patton mentions a *supporting testing term* in addition to the ones explained in Table 2.1.1 [9, p. 15]. It is called the *Product Specification*, *Product Spec* or just *Spec*. The Product Spec is a verbal or written agreement among the software development team, *defining the intended product* in every detail. On the one hand it describes the product itself, how it will work and what it will do. On the other hand, it also explains what the product is not designed to do, which is another important aspect for software testing.

From a tester's perspective the Product Spec represents a very important aspect of the software development and the related software testing process[5]. The main reason for its high relevance is that the developer's idea of what the final product should look like in the end may be completely different from the tester's perception [9, p. 55].

[5]Software development & testing process explained in Section 2.4

2.1.2. When Does a Bug Occur?

According to Patton, bugs arise from one out of *five different reasons* [9, p. 15]. The following List 2.1.2 gives a rough overview of these five sources. Still, the most important consideration to be made is, that *everything that just does not feel right*, is a bug.

Product...

1. ...*does* what the Spec says it *should not*
2. ...*does* what is *not specified*
3. ...*does not* what is *specified*
4. ...*does not* what *should be specified*[6]
5. ...*does not feel right* for any reason[7]

2.1.3. Why Does a Bug Occur?

What is quite interesting, is the fact that most bugs are generally not triggered by programming errors [9, pp. 16-18]. However, the main reason for their occurrence is *usually related to the Product Spec*[8]. Nevertheless, there are still *numerous other reasons* triggering bugs, which are mainly related to the design or to the development of a product.

A general rule of thumb is that *"if you can't say it, you can't do it."*, which perfectly describes one of the most important aspects of today's software development and testing. Many of the various sources of bugs are related to an *insufficient or inadequate Product Spec*, which simply means that the desired product has not been clearly defined. Without a detailed specification, it is almost impossible to verify if the intermediate or final product meets all of its expectations.

Also, List 2.1.3 indicates some additional *common pitfalls* leading to software bugs [9, pp. 17-18]. However, this should not be seen as a complete list, but rather as a rough overview of the things that should be kept in mind.

[6] Catch up on forgotten specifications

[7] Product is slow, difficult to use or to understand...

[8] Product Spec described in Section 2.1.1

- *Product Spec changes*
  Constantly changing
  Not clearly communicated
- *Design errors*
  Triggering programming errors
- *Programming errors*
  Software complexity or poor documentation
  Tight deadlines and pressure
  Dumb mistakes
- Bug *duplicates*
  Multiple bugs with same root cause
- *False positives*
  Thought to be a bug, but is none
- *Test errors*
  Leading to false positives

This listing clearly indicates, that it is not always fully obvious, which failures can be classified as such and which fall under a different category [9, p. 13]. There is always the possibility to detect *duplicates or false positives*, which should be kept in mind when testing.

2.2. Why Testing Software?

The importance of software testing has significantly evolved over time, because of the *growing economic risk* associated with software development[9] [3, pp. 687-689]. The ubiquity of computing has lead to *steadily increasing complexity and costs* which is why software testing has become such a crucial factor within today's development process.

One of the key apects that need to be considered in software testing is, that everyone is fallible, which is why *there will always be bugs* [2, p. 3]. Software testing not only supports *evaluating the acceptability and quality* of a product, but it also assists with *identifying existing bugs*.

Moreover, complex software usually leads to code that is hard to maintain and to extend [6]. However, sophisticated testing approaches like TDD *encourage clean code and a simple software design*, thus increasing the overall confidence in a product. Either such a clean design is enforced from the beginning of the development or it is achieved by *code refactoring*, where an existing design is refined, potentially optimized and tested afterwards.

However, code refactoring can only safely be done if it is supported by reliable testing processes, preventing potential issues that can for instance be introduced by automatic merge features of modern version control systems.

Another important consideration is that software testing *adds additional value* to a product [7, pp. 6-7]. Testing is not only about finding bugs, but also about fixing previously discovered issues thus generally *raising the quality and reliability*. Even products that apparently work fine may still contain errors, which is why testing should never be neglected.

Patton [9, p. 13] mentioned that *hardly any bugs are ever obvious*. This is why there is always the need to write sophisticated software tests which enable developers as well as testers to handle change. The *confidence resulting from software tests* fosters the improvement of a product, because the involved people will notice weaknesses therein [13, p. 185].

[9] Economical aspects described in Section 2.3.2

Furthermore, computer programs should always be as predictable and consistent as possible [7, p. 2]. Generally, *software bugs* can be categorized in two levels. The first one consists of *low-level* impacts leading to user inconvenience and the second of *high-level* bugs which lead to financial loss. Additionally, high-level bugs can even be dangerous or harmful, which is why they have to be prevented or fixed.

2.3. Important Aspects of Software Testing

Osherove [13, pp. 151-152] makes clear that software tests are only of little value if not all of the *three highly relevant attributes* do apply, which are summarized within the following paragraphs.

Trustworthiness Test engineers should be confident that all tests *verify the predetermined things* mentioned in Section 2.1.2, while *remaining bug-free* themselves [13, p. 151]. If the according trust exists, it is easy to accept a test result, independent from whether it indicates a success or a bug.

Maintainability Unmaintainable tests may *ruin schedules* or *get sidelined* in case of very tight deadlines [13, p. 151]. As the maintaining of such tests requires a *vast amount of time and money*[10], developers might stop supporting them if the tests change too frequently.

Readability This aspect is *tightly connected* to the other two key aspects mentioned before [13, pp. 151-152]. Generally, there is just *no way to trust a test if it cannot be read*. Additionally, if a test is unreadable, it also becomes *more difficult to maintain* it, because it cannot be fully understood.

This is why it is extremely important to know that "people who can read your tests can understand them and maintain them, and they also trust the tests when they pass." [13, pp. 184-185].

Another important consideration to be made is that software tests usually *grow and evolve with the tested product* [13, pp. 184-185]. Therefore, professional software testers should always strive to adhere to good tests while trying to get rid of bad tests, either by removing or optimizing them.

Without any doubt, software testing is a technical task [7, p. 5]. Nevertheless, it also involves many relevant *psychological and economical aspects*, which are described in the following sections.

[10] Economical aspects described in Section 2.3.2

2.3.1. Psychological Aspects

In general, most software developers and testers have an individual testing model in mind, because software testing is as old as the development of software itself [3, p. 687]. For this reason, *various definitions of the success* of testing exist, including its scope and corresponding objectives.

Nevertheless, the definition and the according mental model of testing is a key factor of successful software testing [9, p. 19]. Software testers should *not aim to confirm that a tested product works* perfectly, but their goal should be to *find bugs*. Although these two definitions of successful testing seem to be quite similar, there is a big distinction from a psychological point of view. If testers mainly focus on verifying that a certain software product works as expected, they usually tend to configure the tests in order to succeed while missing critical bugs.

Myers, Sandler, and Badgett suggest to assume that every *software does contain bugs* [7, pp. 5-7]. This is why it should be every tester's main motivation to *find and fix* as many errors as possible, and not to proof that the product works as expected. Generally, human beings are known to perform weakly when trying to solve infeasible or even impossible tasks. For this reason, the correct definition of the purpose of testing is crucial, because it circumvents this critical psychological flaw.

Moreover, this definition of successful testing is especially important, because people are generally *goal-oriented*. Therefore, *good testers* tend to have a rather destructive outlook in comparison to the developers creating the product. Nevertheless, such a *destructive mindset* is one of the key tools in order to enforce the observation of undiscovered bugs.

Additionally, there are other important factors that can be related to successful testing [7, pp. 5-7]. According to the correct definition of a successful test, it should aim to discover bugs. Therefore, a test case should only be described as being *successful if it finds errors that can be fixed*. In case there are no errors, a good test case would also indicate that *no more bugs* can be found. Especially this description often gets misinterpreted, because many people call it a successful test run if no bugs have been found [3, p. 691].

2.3.2. Economical Aspects

In addition to the psychological aspects mentioned in Section 2.3.1, economical aspects play another very important role in software testing [7, p. 5]. Generally, it is economically *infeasible to test a product completely*, because it would just require too many resources and therefore take too much time. Nowadays, software development teams spend about *50% of the time* on testing, resulting in *more than 50% of the costs* in total [7, p. IX].

Figure 2.1 indicates that time plays a key role in software testing, because *discovering bugs sooner makes them cheaper to fix* [9, p. 19]. This is why testers should always ask themselves how to optimize the testing process[11].

Myers, Sandler, and Badgett [7, p. 20] also clearly stress the fact that testers should aim at finding bugs as early as possible. This way the costs of correcting earlier observed bugs are much lower. Additionally, a sophisticated solution for a specific problem is generally easier to find if it is observed sooner in the development process[12].

As already mentioned, it is usually not possible or just impractical to aim at finding all existing errors [7, p. 8]. This is why software testers have to make assumptions about the software in order to define a proper test case design. Resulting from these assumptions, sophisticated *testing strategies* have to be established. This factor will be discussed more thoroughly in Section 2.4.2.

[11] Testing process described in Section 2.4.2
[12] Software development process described in Section 2.4

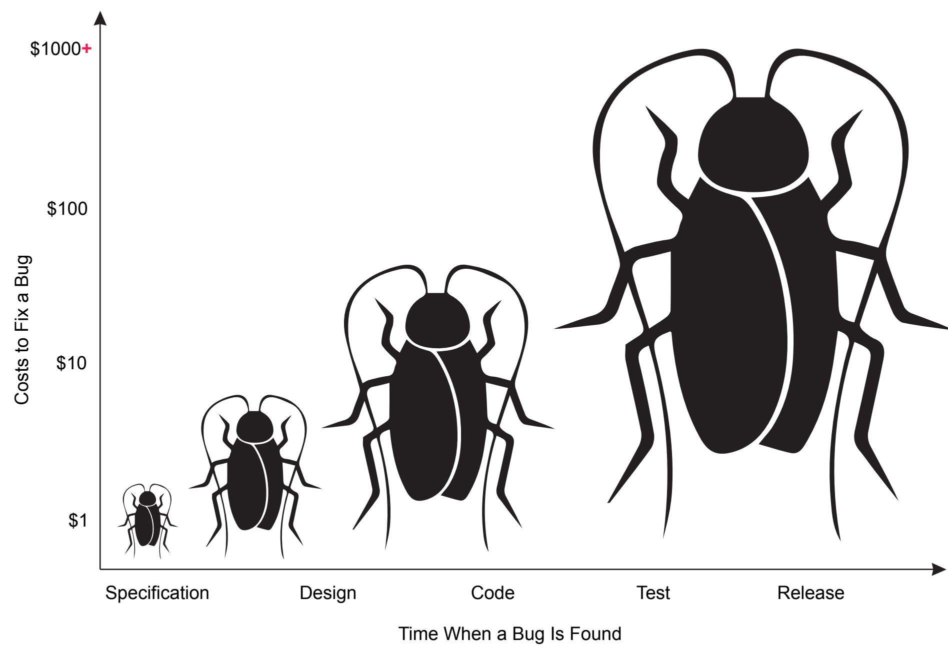

Figure 2.1.: The costs to fix a bug increase exponentially over time. For this reason, it is important to discover and fix a bug as early as possible. [9, Image adapted from p. 18]

2.4. Software Development Process

Software development is usually a *planned and well-structured process* [9, p. 18]. It starts with the initial idea, continues with planning, development and testing. Still, each of these steps potentially introduces new bugs.

The development of software may only involve few, but it can include hundreds or even thousands of people [9, p. 24]. If a lot of participants are engaged with developing a product, there have to be more or less strictly *separated roles and phases* in order to work together under potentially tight deadlines. The development process defines what the individual members do, how they interact and how they make decisions.

2.4.1. Development Lifecycle Models

Software lifecycle models primarily *define the stages* and the corresponding *order of stages* concerning software development [14, p. 345]. Additionally, the *transition criteria*[13] between the stages gets predefined as well.

The following *two main questions* are addressed by every single of all the different lifecycle models [14, p. 346].

1. What to do *next*?
2. *How long* to continue?

This section provides a rough overview about the main – more or less structured – approaches defining the creation process of a software product [9, pp. 30-31]. All these different models range from the initial conception to the final public release of a product. However, there is *no solution* that fits perfectly and would work *for any kind of software project*.

In the following subsections the *four most common models* are explained [9, pp. 30-36]. Although *better-known approaches* exist, these can be considered mostly *only as variations* of these four. Still, it is always important to consider the current requirements of a certain project and to tailor the according testing process to the deployed development process[14].

[13] Transition includes completion criteria for current & entrance criteria for next stage
[14] Testing process described in Section 2.4.2

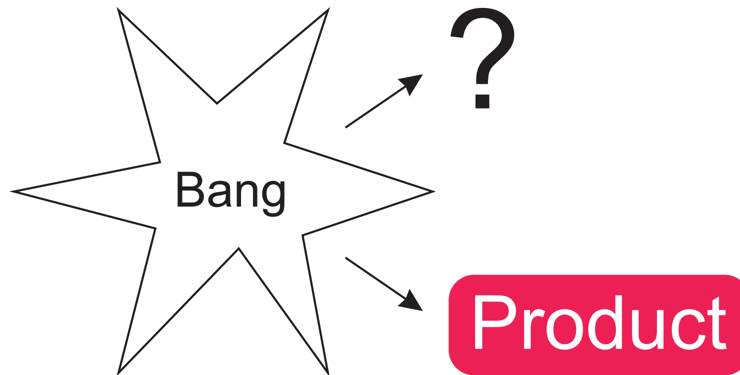

Figure 2.2.: The Big-Bang Model is the simplest software development method. Either all the effort put into it results in the desired product or it does not, where all resources have just been wasted. [9, Image adapted from p. 31]

Big-Bang Model

The Big-Bang Model shown in Figure 2.2 describes the *simplest software development lifecycle* [9, pp. 31-32]. Although this approach is so simple, it still takes a lot of resources and effort to build a product[15].

There is only *little planning and scheduling* involved and the development process stays *informal* [9, pp. 31-32]. For this reason, there is no confidence or certainty that this development model will result in the desired product, because it is *never clear what to expect* or how the specifications look like.

This is why the Big-Bang Model usually gets adopted if the according *product requirements are not really understood*.

Additionally, the corresponding *customers need to be flexible*, because they will not know what to expect until the very end, when the product gets finally launched [9, pp. 31-32].

[15]Big-Bang Model requires many people, time & money

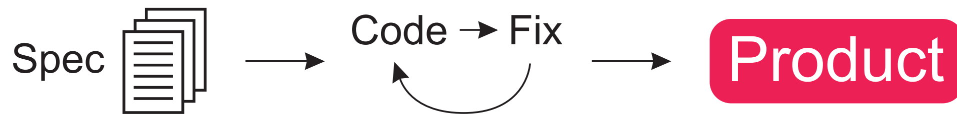

Figure 2.3.: The Code-and-Fix Model of software development contains a cycle of coding and fixing, which is repeated until someone decides it should stop. [9, Image adapted from p. 32]

Code-and-Fix Model

Usually, the Code-and-Fix Model is used if there is no specific reason to use another development lifecycle model instead [9, pp. 32-33]. In contrast to the Big-Bang Model[16], with this approach one at least needs a small amount of *knowledge about the product requirements*.

Nevertheless, this lifecycle model describes yet another very *code-driven* way to build a product, because the involved people first code and think about all the other requirements later[17] [14, p. 346].

"There's never time to do it right, but there's always time to do it over" [9, p. 32]. This quotation of Patton perfectly summarized the essentials of the Code-and-Fix Model, which are also abstractly illustrated in Figure 2.3.

Generally, a project team using this approach starts with a basic idea and design, and later continues to *iterate with development, testing and bug-fixing* [9, pp. 32-33]. However, this repeating cycle can take some time until the team finally decides to release the product.

The Code-and-Fix Model is *well-suited for smaller projects*, because it requires only very *little planning and documentation* [9, pp. 32-33]. Additionally, it is very easy to quickly present results by creating prototypes or samples of a product. Moreover, this software development approach has even been *used by largely adopted and successful products*.

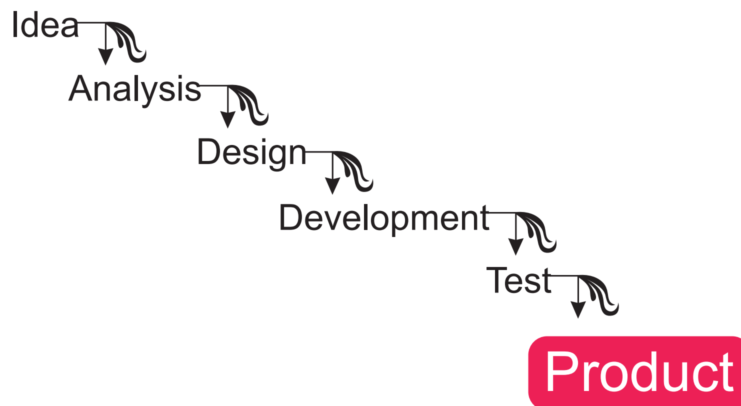

Figure 2.4.: The Waterfall Model defines a software development process flow from one distinctive level to the next one. If this approach is strictly implemented, there is no way to go back to a previous step. [9, Image adapted from p. 33]

Waterfall Model

The Waterfall Model has been around since the beginnings of software development and is usually also the *first one to be taught* when becoming a software developer [9, pp. 33-34]. Patton describes it as a *simple and elegant* solution, which just makes sense.

Due to the recognition of some drawbacks concerning the software life-cycle models already mentioned[18], the Waterfall Model was introduced, providing *successive stages* for the development process [14, pp. 346-347].

Figure 2.4 clearly indicates the basic flow of the Waterfall Model of software development [9, pp. 33-34]. It specifies several *independent steps*, starting from the initial idea to the resulting product.

Moreover, an important aspect is that the *project team reviews* the current project state at the end of *each step*, deciding when it is time to continue with the next level [9, pp. 33-34].

[16] Big-Bang Model described in Section 2.4.1
[17] Other requirements like design, testing & maintenance…
[18] The Big-Bang & Code-and-Fix Model described in Section 2.4.1

Additionally, Patton also indicates the *three most important factors* considering the Waterfall Model, which are described in List 2.4.1 [9, p. 34].

1. Detailed *Product Spec*
2. *No overlapping* steps
3. *No way back* to previous[19]

Especially the aspect that there is *no way back* to a previous step is *limiting*, because it requires a highly professional and disciplined project team to constantly hold on to the predefined development flow [9, p. 34].

For this reason the team has to *define every aspect of the product before starting* to write the first line of code. The problem is that the requirements may already have changed until then. Especially this *inflexibility* illustrates one of the main disadvantages of the static process of the Waterfall Model.

Nevertheless, nowadays fortunately *some variations* of the Waterfall Model exist, which loosen the rules so a certain degree of *step-overlapping* while additionally providing *ways to step back* is accepted [14, p. 347].

This loosening of the predefined development lifecycle could for example be achieved by introducing *feedback loops* between stages or by an initial adaption of *software prototyping* in parallel to the analysis and design phase.

However, even with all these possible refinements, there is no way to eliminate all the *fundamental issues* related to the Waterfall Model[20] [14, p. 348]. This is the reason why alternative development models like the Spiral Model[21] were evolved, which aim to avoid these critical problems.

[19] Strict Waterfall Model does not allow to go back to previous step
[20] Waterfall Model requires detailed Spec as completion criteria
[21] Spiral Model described in Section 2.4.1

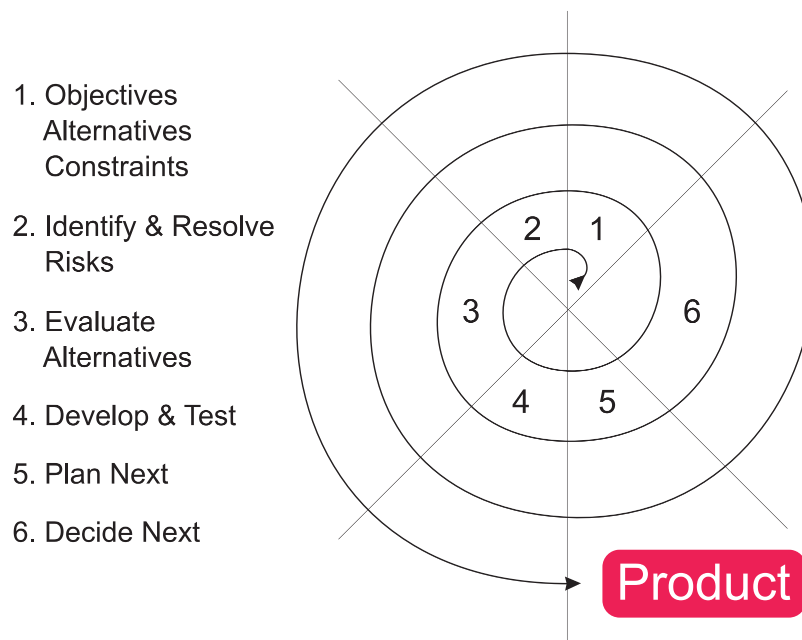

Figure 2.5.: The Spiral Model starts very small and continually expands by iterating through predefined software development phases. This is why, the product becomes more defined and stable over time. [9, Image adapted from p. 35]

Spiral Model

The Spiral Model of software development was introduced by Boehm in 1986 [15]. It does not only address some of the main issues inherent with the models explained in the sections before, but it additionally *includes some major improvements*[22] [9, pp. 34-35].

The most noteworthy differentiation of the Spiral Model is, that it is a rather *risk-driven* software development approach opposed to the other more document-driven or code-driven variants [14, p. 345].

Figure 2.5 abstractly illustrates the basics of the spiral software development process [9, p. 35]. An important aspect is that *not every detail* has to be specified in the *first stage* of the product development lifecycle.

[22]Spiral Model improvement examples shown in List 2.4.1

The development team builds up the product in many *iterations* similar to a spiral, which *starts small and continues to grow* [9, p. 35]. This is why not every detail already gets defined in the first place. The team starts small, decides about important features, builds and tries them, asks for the customer's feedback and moves on to the next level afterwards. This *iterative process* is repeated until the team decides to launch the product.

Figure 2.5 illustrates all *six steps* around the spiral thus providing a basic overview [9, p. 36]. Obviously, *many similarities* to the models already mentioned can be observed. The six spiral steps can be compared to the Waterfall Model[23], the spiral itself corresponds to the Code-and-Fix Model, while the look from the outside brings the Big-Bang Model back to mind.

The Spiral Model provides various *additional advantages* in comparison to the already illustrated lifecycle approaches [14, p. 359]. List 2.4.1 indicates few examples to provide a deeper understanding regarding the potential of the Spiral Model incorporation.

- Early focus on *options* & *alternatives*[24]
- Prepares for *evolution* & *growth* of product[25]
- Enables to include *quality objectives*[26]
- Eliminate errors & useless alternatives early[27]

However, there are generally *no software projects that flawlessly follow* the specified principles and steps of this lifecycle model [9, p. 38]. The reason for this is that the team members are usually *never given a perfectly detailed Product Spec*, which meets all costumer's expectations and desires.

Nevertheless, it is still quite essential to know how the ideal software development process should look like in order to have something to aim for [9, p. 38]. Generally, there are always compromises to be made, because a sophisticated software development process should be tailored to the needs of a specific project.

[23] Analysis, design, develop & test like in the Waterfall Model of Section 2.4.1
[24] Spiral Model simplifies consideration & reuse of existing software
[25] Objectives as major source of product change
[26] Identify various objectives & constraints
[27] Risk analysis & validation steps eliminate alternatives

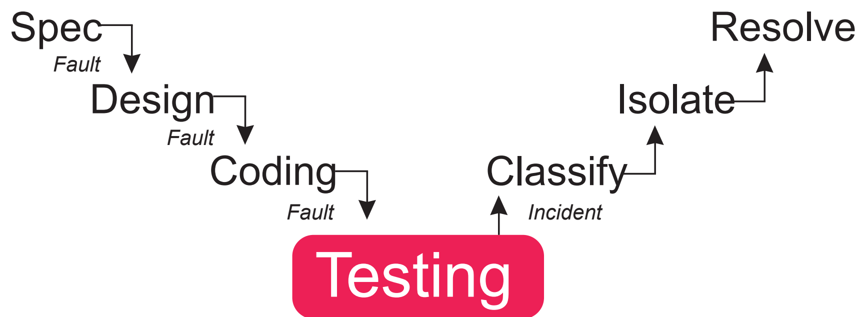

Figure 2.6.: A lifecycle model for testing implies three opportunities for bugs to arise. Bugs may be introduced within the Spec, the design or the coding phase, which are propagated through the whole product lifecycle. [2, Image adapted from p. 4]

2.4.2. Software Testing Process

Software testing is not a single step, but rather a process on its own, which *colludes with the development process* of a product [7, p. 2]. The understanding of the importance of how to make software testing effective has grown over time, caused by an evolution of the according process models [3, p. 687].

Figure 2.6 illustrates the basic scheme of a testing lifecycle, where all of the development phases provide new opportunities for bugs to be introduced [2, p. 4]. The faults planted in one phase usually propagate through all the subsequent phases as well. Another important consideration is that also a potential resolution of a fault may lead to new issues.

The *process of testing* itself can be subdivided into *four steps*, which are summarized in List 2.4.2 [2, p. 4]. The steps named "Creation" and "Execution" are related to the construction and the execution of all predefined test cases resulting from the test planning stage.

1. *Planning*
2. *Creation*
3. *Execution*
4. *Evaluation*

Myers, Sandler, and Badgett point out, likewise, that a sophisticated testing process must include a *thorough inspection of the results* [7, p. 15]. Although this should be obvious, it is still one of the main sources of missing clearly indicated bugs. As already mentioned in Section 2.3.2, software testers should always aim at finding bugs as early as possible. If this is not the case, it gets increasingly more expensive to fix a bug later on.

Testing with Lifecycle Models

The following sections will roughly address the basic principles of the individual testing processes corresponding to the development lifecycle models mentioned in Section 2.4.1.

Big-Bang Model Concerning the Big-Bang Model[28], there is only *little formal testing* involved [9, p. 32]. If any testing happens at all, it is usually only done *at the very end*, right before launching the final product.

Nevertheless, the big advantage considering the testing of a software with the Big-Bang Model is that the *product is already complete* when testing starts [9, p. 32]. Such a complete product can be considered the *perfect test specification*, because there are no uncertainties that need to be thought of.

However, this advantage of having a complete product as a test specification might also highlight a more *negative aspect*, because the dilemma is that the tester's job is mostly just to *report bugs instead of fixing* them directly. It becomes very hard to do that, because the product has to get launched as soon as possible. Therefore, software testing is unfortunately only viewed as something that keeps the product from its final delivery.

[28] Big-Bang Model described in Section 2.4.1

Code-and-Fix Model Although the software testing procedure is not specifically mentioned within the name of the Code-and-Fix Model[29], it plays a *significant role between coding and fixing* [9, p. 33].

The central part of the Code-and-Fix testing process is defined by a *cycle of running tests, reporting bugs and continuing* with the next test round. Because of this, there is always the possibility that the testing of one round has not been finished yet, when already the next product release is ready for testing. In the meantime, the Product Spec or features may have changed, which would require to adapt the tests accordingly. This is why software testers should always be aware of this and keep it in mind during the testing.

There are *three main disadvantages* regarding the Code-and-Fix Model, which are mentioned in the following listing [14, p. 346].

1. Often do *not meet requirements*
2. *Hard to test* & *modify*
3. Subsequent *fixes expensive*[30]

Waterfall Model From the perspective of using the Waterfall Model as a software development lifecycle[31], testing is usually *straightforward*, because everything has already been carefully *specified beforehand* [9, p. 34].

Before the testing process starts, every detail has been defined and has already turned into the current state of the product. This is why software testers should *know exactly what to test* and there is no question whether something is a feature or an issue.

On the other hand, testing *at the very end* of the development process is also the *main disadvantage* when using the Waterfall Model [9, p. 34]. A certain bug could potentially be introduced in an early stage, without being detected until the very end, which is the most *expensive way of a bug-observation*[32].

[29] Code-and-Fix Model described in Section 2.4.1
[30] Code-and-Fix Model unstructured & hard to maintain
[31] Waterfall Model described in Section 2.4.1
[32] Importance of early bug-detection described in Section 2.3.2

Spiral Model A software tester, who is involved in a spiral software development lifecycle[33] has the opportunity to *influence the product* in an early stage of the product development[34] [9, p. 36].

For this reason, software testers should know how the current product has evolved and where it is heading to in the future. Another big advantage is that *no extreme time pressure* before launching the final product is existent, because not all testing has to be performed at the very end. Therefore, the overall costs of finding bugs are lower[32], which is an important aspect of a sophisticated testing process.

Who Should Test Software?

According to Osherove [13, p. 184], good testers need *discipline and imagination* to excel at software testing. Furthermore, Jorgensen [2, p. 435] emphasizes this even more by mentioning additional key attributes like *ingenuity, curiosity* and a *"can-I-break-it mentality"*.

This very *destructive mentality*[35] is also highlighted by Myers, Sandler, and Badgett [7, pp. 5-7]. However, this is usually the opposite to the perspective of a software engineer, who wants to create something rather than destroy it. For this reason, the developers of a certain product are usually not the best people to consult for testing purposes[36].

As already explained in Section 2.4.2, it should be every tester's goal to find bugs with the intent to discover them as soon as possible [9, p. 19]. In addition to finding bugs, testers also have to make sure all the observed issues get *documented and fixed* as well.

Another important aspect is that software testers generally act as the *customer's eyes*, because they are the first to try a certain product [9, p. 19]. Because of this test engineers must *demand perfection*, even if not all of their requests may get fulfilled.

[33]Spiral Model described in Section 2.4.1
[34]Tester involved in preliminary Spiral Model design phases
[35]Destructive, even sadistic outlook of good software testers
[36]Not test own product principle described in Paragraph 2.4.2

Obviously, some of the major testing principles are also tightly connected to the question of who should test software[37] [7, p. 13]. Two of the most critical aspects are summarized in further detail within the next paragraphs.

Not Test Own Product As already mentioned, the testing of a product should be done by someone who has *not been involved in the product development* [7, pp. 14-15]. One of the reasons for this is that bugs could occur due to an misunderstanding of the initial specification, which would lead to a misinterpretation of the test requirements. In addition, if someone knows how a certain product works, it is hard to detect any hidden bugs. Last but not least, no one wants to discover errors which may have been introduced by the observer in the first place.

Furthermore, the *organization* itself should *not test its own* products as well [7, p. 15]. An organization faces *similar psychological problems*[38] as an individual regarding software testing. Additionally, companies are usually largely evaluated on their ability to deliver by a given date while keeping the costs low[39]. So, opposed to time and money, it is *hard to quantify the reliability* of a product. This is why software testing is often perceived as an annoying factor which should get limited in order to reach predefined schedules or cost objectives.

More Bugs to Find Professional software testers must be aware of the fact that usually *always more bugs* can be found [7, pp. 16-17]. Even after extensively testing the whole product without finding any more issues, it is generally safe to assume that there are still more errors to be detected. Although this key aspect of successful software testing was already mentioned before[38], it has to be emphasized once more.

Nevertheless, these general guidelines must still not be applied in all possible circumstances [6]. For example, if a product gets developed using TDD, all resulting software tests are developed using a *"test-first" approach*, which is why they must of course be *created by the responsible software engineer*, because the tests are the foundation for the subsequent development.

[37] Testing principles described in Section 2.6
[38] Psychological aspects described in Section 2.3.1
[39] Economical aspects described in Section 2.3.2

Additionally, software developers may also be important test team members, especially for *code evaluations, debugging* and the *isolation of bugs* [7, p. 15]. Finally, observed and isolated bugs usually should get *fixed as well*, which is also the developer's job in an ideal world.

When Testing Software?

As already mentioned in Section 2.3.2, it is desirable to discover and fix bugs *as early as possible* [9, pp. 53-55]. This is why the *Product Spec* does not only describe the intended product, but it serves as the *perfect test specification* as well[40]. Therefore, the Product Spec should also be considered as a testable item in order to observe bugs even *before writing the first line of code*. Unfortunately, it is usually not this simple for software testers to test the Product Spec itself and therefore the discovery of bugs gets delayed[41].

Section 2.4.1 clearly explaines why the testing process is tightly connected to the established software development model. This is why it is crucial to choose the right development approach beforehand, in order to allow sophisticated testing procedures to be implemented.

"I love deadlines. I especially like the whooshing sound they make as they go flying by." by Adams [16]. This quote highlights one of the main issues related to software testing today [7, p. 20]. An organization's *internal pressure* may be the main source for a miserable testing culture. Statements like "Fix this bug as fast as possible" are unfortunately quite common nowadays. Therefore, software testers are often under a remarkably high pressure. This is yet another reason why it is so important to choose an appropriate development model, because it reduces the pressure through an early incorporation of testing methodologies.

[40] Product Spec described in Section 2.1.1

[41] Testers might join when product almost or completely finished

2.5. Software Testing Levels

What is very essential is that sophisticated *testing strategies* have to be established before starting to test [7, p. 8]. Nonetheless, it is important to select the *right strategy at the right time* thus making testing efficient and effective [8, p. 213]. Two high-level methods of testing are named *Black-Box* & *White-Box Testing*, which will be explained in the following sections.

These two terms are commonly used to describe today's most relevant testing levels. Every single level has to precisely *specify all objectives*, which have to be monitored throughout the testing process [2, p. 435].

According to Jorgensen, an application usually requires *at least* two testing levels, named *Unit Testing*[42] *and System Testing*[43] [2, p. 435]. Furthermore, the level of *Integration Testing* turns out to be well-suited for *larger software*.

2.5.1. Black-Box Testing

Myers, Sandler, and Badgett describe the approach of Black-Box Testing as a *data-driven or input/output-driven* approach [7, pp. 8-10]. Software testers are not concerned about the composition or the internal behavior of a product, but they solely consider how the software under test should behave [9, p. 55]. Obviously, this is where the name originates, because it is impossible to look inside the "black box".

For this reason, testers aim at verifying that a product behaves as required by *testing the inputs and expected outputs* [7, pp. 8-10].

However, it is unthinkable to test every possible input combination, which is known as *exhaustive input testing*[44]. This is why Black-Box Testing cannot guarantee a program to be completely error-free. Furthermore, the testing investment's outcome has to be maximized by trying to *discover a maximum* amount of bugs *with a limited number of tests*[45].

[42] Unit Testing described in Section 2.5.3
[43] System Testing described in Section 2.5.3
[44] Exhaustive input testing economically infeasible
[45] Maximize output of testing investment relating to Section 2.3.2

2.5.2. White-Box Testing

In opposite to Black-Box Testing, White-Box Testing[46] allows and even requires to make reasonable *assumptions about the program* under test [7, pp. 10-12]. Therefore, this form of *logic-driven* testing considers the internal structure by inferring data from the program's logic[47].

Patton emphasizes this by explaining that testers using a White-Box Testing approach have *access to the program's code* which provides helpful testing clues and simplifies to tailor the testing according to the needs of the project [9, p. 56]. Because of this, *programming know-how* is essential too, because it enables testers to understand how the program's code works [8, p. 214].

Nevertheless, White-Box Testing approaches still face *similar drawbacks* as Black-Box Testing alternatives [7, pp. 10-12]. The reason for this is the overall goal to execute all program paths at least once[48].

Similar to the exhaustive input-testing of Black-Box Testing, this form uses *exhaustive path-testing* that is also highly *inadequate or impossible* to accomplish. Furthermore, even if all paths have been covered, the tested program could *still contain errors*, because of the *unanswerable questions* summarized in List 2.5.2.

- Product *Spec fulfilled*?
- *Absence of paths*?
- *Data-sensitivity* issues?

Although exhaustive input testing is generally more practical than exhaustive path testing, none of these approaches are economically feasible if established individually [7, p. 12]. For this reason, it is usually the *best to combine Black-Box and White-Box Testing* in order to deploy a sophisticated testing strategy.

[46] White-Box Testing sometimes called clear-box testing
[47] White-Box Testing derives data, often at neglect of Product Spec
[48] Program path is one possible way of code execution

2.5.3. Test Level Subdivision

Software testing levels relate to different subsets of a program under test [17, p. 136]. Usually, different program scopes get tested in a predefined order, thus providing the *scheme and sequence* for the overall testing process.

In general, software testing is subdivided into *four distinct levels of testing*[49] [8, p. 213]. The reason for this differentiation is that all of these levels serve individual purposes, which are briefly summarized within this section.

Unit Testing

Unit Testing is considered to be the *initial testing step*, focussing on the smallest building blocks of a program [7, pp. 85-116]. It aims at identifying discrepancies between a program's module and its specification[50].

Although there are various definitions available, a *unit* can generally be described as the *smallest testable part* of a program [8, p. 213]. Such a unit usually relies on one or *few inputs*, while providing only a *single output*.

Myers, Sandler, and Badgett mention *three major motivations* for Unit Testing, which are summarized in List 2.5.3 [7, p. 85]. It is important to indicate that the other test level subdivisions of Integration and System Testing are building up on top of these considerations.

1. Allow *combined testing*
2. *Simplify debugging*[51]
3. Introduce *parallelism*

Unit Testing is done *by software developers* who have the *required insight and knowledge* to allow testing of individual parts of the software [8, p. 213].

Nonetheless, the Unit Testing level alone is not enough, because only a form of higher-order software testing enables the observation of otherwise non-observable bugs [7, p. 113].

[49] Four distinct levels of Unit, Integration, System & Acceptance Testing
[50] Unit Testing also known as module testing
[51] Detect bug located in a particular module

Integration Testing

According to Myers, Sandler, and Badgett, the development of software represents a process of communicating and translating information [7, p. 113]. This is why it is simply not enough to test solely the smallest, individual parts of a software[52], but they have to be *tested during interaction*.

Usually, Integration Testing is done *after Unit Testing* [8, p. 214]. Still, opposite to Unit Testing, the intent of Integration Testing is to verify that a *group of units* is working together as expected. Therefore, the overall goal is to expose a multitude of faults concerning the *interaction between units*.

Integration Testing is either done by the *developers* themselves or by other completely *independent testers* [8, p. 214].

Nevertheless, Jorgensen mentions that the purpose of Integration Testing is generally not well-understood, because it is often poorly realized in practice [2, p. 229].

System Testing

In general, the level of System Testing is the next logical testing step *after Integration Testing* [7, pp. 119-120], [8, p. 214]. Although this can be seen as the testing process closest to an everyday experience, it is probably the *most misunderstood and most difficult* one [2, p. 253]. As the name would suggest, System Testing does not simply mean to test the functionality of a system. However, the real purpose of this form of testing is to highlight the discrepancies between a program and its initial objectives, thus evaluating the *system's compliance to its specification*.

For this reason, System Testing focuses on *translation errors* made within the Product Spec creation process. Therefore it typically tends to be error-prone, which is why the level of System Testing is such an essential element of every sophisticated testing strategy.

System Testing is usually done solely by *independent testers* who are not involved in the according software development process [8, p. 214].

[52]Unit Testing described in Section 2.5.3

Acceptance Testing

Acceptance Testing is generally concerned with the *acceptability of a system* [8, p. 214]. In opposite to System Testing, it verifies the compliance of a system to its *business requirements*, not comparing it to the Product Specification.

Because of this, Acceptance Testing aims at verifying a program's compatibility to its *initial requirements* and current *user needs* [7, pp. 131-132].

Therefore, Acceptance Testing evaluates whether a product is acceptable and ready for its final delivery. Furthermore, *no strict testing procedures* must be adhered in Acceptance Testing, but testers usually follow an ad-hoc software testing approach.

However, Acceptance Testing is a rather unusual testing type, because it mostly is not done by software developers or testers, but it is *performed by the product's customers or end users* [7, p. 131].

2.6. Software Testing Principles

Patton describes these principles as the "rules of the road" or "facts for live" for any software engineer or tester [9, p. 38].

Although, the guidelines firmly summarized in this section seem to be quite obvious, they still get *overlooked too often* [7, p. 12]. Nonetheless, all of the illustrated testing protocols result from a stripped-down combination of the ones mentioned within the books *The Art of Software Testing* [7, pp. 12-18] and *Software Testing* [9, pp. 38-43].

Not Test Own Product It is usually a bad idea to test one's own product regardless of whether it concerns individuals or a whole organization[53], which is generally due to various *psychological reasons*[54] [7].

Input & Expected Result Tests must define the input and correct output [7]. *Not only valid inputs* must be checked, but also *invalid or unexpected* ones. The results must be carefully inspected in order to not miss any bugs.

More Bugs to Find There is no way to verify a program is bug-free [7], [9]. Therefore, a test can *show a bug exists, but it cannot do otherwise*. Testers have to assume there are still bugs which have not been found yet[55].

Bugs Follow Bugs Bugs tend to *come in clusters*, which is why some parts are more error-prone than others[56] [7], [9]. If software testing fails to discover more bugs, there are only few or even none left to find.

Impossible to Test All It is usually not possible to test a certain product completely[57] [9]. There are just *too many inputs/outputs and paths* available. Additionally, the Product Spec and according *bugs are subjective*.

[53] Not test own product described in Section 2.4.2
[54] Psychological aspects described in Section 2.3.1
[55] Correct test assumption/definition described in Section 2.3.1
[56] Testing should focus on parts more prone to bugs
[57] Economically infeasible to test all described in Section 2.3.2

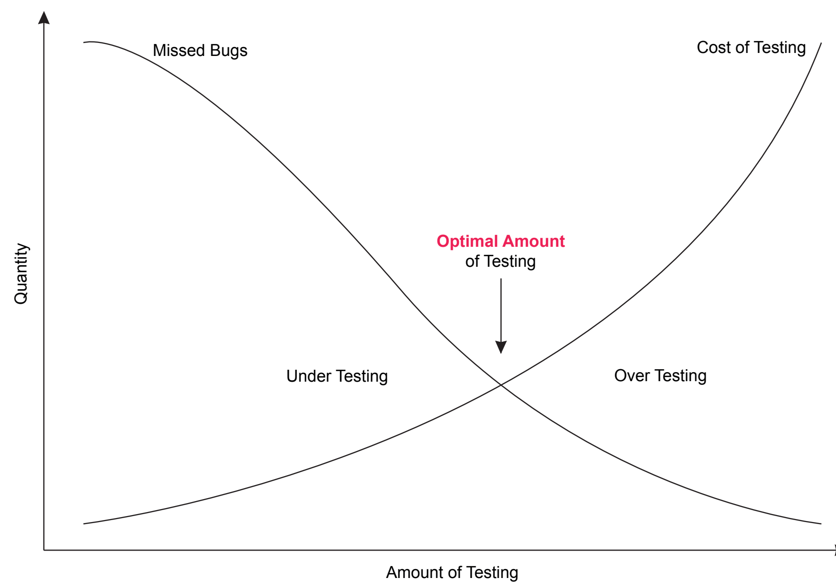

Figure 2.7.: This graph indicates the relationship between the test effort and the observed number of bugs. It clearly illustrates that "testing, like everything else, can be either underdone or overdone" [3, p. 688]. [9, Image adapted from p. 40]

Risk-Based Task Testers have to take the *risk of missing bugs* [9]. Therefore, risk-based decisions have to be made about what and how much should be tested[57]. Figure 2.7 illustrates the *optimal amount* of software testing.

Creative & Intellectual Challenge Testing is challenging regarding the creativity and intelligence [7]. Although there are certain methodologies, it still requires a lot of *creativity and experience to observe nontrivial bugs*[58].

"Pesticide Paradox" Introduced in 1990, clarifying that more *testing leads to an immunity of programs* [9], [18]. Multiple reruns expose all observable bugs. Therefore, new tests have to be created to discover more issues.

[58]Creative & intellectual challenge described in Section 2.4.2

No Throwaway Tests Testing represents a *valuable investment*[59] [7]. Therefore, reinventing tests frequently must be avoided[60]. Test cases should be saved in order to run them again later, which is known as *regression testing*.

Not All Bugs Fixed Not every bug gets fixed, because perfection is not desirable[59] [9]. *Trade-offs* have to be made that *decide which bugs to fix*. The reason for this can be time-pressure, misunderstandings or high risk[61].

In addition to these main principles, there are more *specific testing guidelines* available at `http://www.softwaretestingstandard.org/`, providing a very detailed overview regarding test organization and processes [19]. However, these guidelines represent an *internationally agreed set of principles*, which constitute a solid foundation for software testing.

[59] Economical aspects described in Section 2.3.2
[60] Test reinvention requires a lot of resources
[61] Bug-fixing risk to introduce even more critical errors

3. UI Testing

The intent of this chapter is to explain the required basics of User Interface (UI) Testing and UI Test Automation, including a motivational section discussing why an UI should be tested. Its main goal, however, is to provide a summary of all major considerations, guidelines and tools, in general as well as specificly to the iOS development platform.

Still, this thesis' main intent is to focus on *UI Testing in iOS*, which involves testing of mobile devices providing Graphical User Interfaces (GUIs). This is why the two terms of GUI Testing[1] and UI Testing are used interchangeably.

3.1. What Is UI Testing?

In general, UI Testing can be described as a *System Testing* approach[2] focussing on the *GUI perspective* of an application[1] [20, p. 55]. Additionally, it is also used for *Acceptance Testing*, which means verifying the application behavior through the user's eyes [21, p. 11]. This is why UI Testing concentrates on the program's behavior including *device specific* functionality like gesture interactions [22, p. 50]. Important to note is that UI Testing is either done *manually* or *automatically* by using special automation tools[3] [23, p. 10].

According to this definition, performing interactions on a program's GUI marks the foundation of UI Testing [23, p. 10]. User interactions are encoded as *event sequences*, which is a test requirement next to the *input data*. Event sequences are performed in a certain order when testing, which gets predefined by an application *specific GUI Model*[4] [24, p. 1479].

[1]GUI described in Section 3.1.1
[2]System Testing described in Section 2.5.3
[3]UI Test Automation described in Section 3.5
[4]GUI Model described in Section 3.1.1

Moreover, UI Testing is often connected to the software's logic, because there is usually the need to *analyse runtime properties* while executing tests [25, pp. 204-205]. This is due to the overall goal of navigating to every available GUI page, while analyzing the *internal states and outputs*.

Nonetheless, as *countless input possibilities* exist, the number of event combinations is infinite [23, p. 10]. This is why, UI Testing alone cannot cover all possible scenarios of a program's user interaction.

3.1.1. Definition of Terms

The purpose of this section is to provide an overview of the most important terms related to the matter of UI Testing. These terms are used within the succeeding sections to describe the process of UI Testing in further detail.

Graphical User Interface (GUI) Graphical User Interfaces are the most common *way to interact with a program* today [23, p. 10]. GUIs represent a *collection of widgets* like buttons or text boxes[5], providing a familiar way for users to interact with. Furthermore, user interactions *trigger events* which can be evaluated and handled by an application.

GUI Model A GUI Model's purpose is to illustrate the *connected structure* of a GUI [26]. Generally, it consists of *states and transitions* between states, where a *state* summarizes actions, while a *transaction* indicates moving from one state to another. Such a transaction is triggered by a user interaction event, which gets handled by the tested program.

Code Coverage Code Coverage is used to *measure the effectiveness and quality* of a test case[6] [20, pp. 55-56]. It is determined at statement, branch or path level, thus evaluating the coverage of a *business logic*. Nevertheless, this kind of quality assessment is based on the program's *assumptive determinism*.

[5]Widget is a graphical element of a GUI
[6]Fault detection is another important test metric

Flakiness A test is called "flaky" when it is *impossible to repeat the exact same execution* in a reliable way [24, p. 1479]. So, another test run may lead to *different results* although neither the program nor the test have changed [20, p. 56]. Generally, Flakiness does not only affect test results, but also the experimentation on new approaches.

Invariant An Invariant is a *high-level abstraction* of the underlying program's implementation [20, p. 56]. However, as Invariants represent properties that are valid for all test cases, they *should be consistent* between test runs in order to reduce the Flakiness of a test.

3.1.2. UI Testing Categories

There are *three main categories* of UI Testing, which are shortly summarized within this section [26]. These categories range from completely random testing to highly sophisticated learning-based UI Testing.

Random UI Testing The purpose of this category is to discover bugs by *randomly firing events* on a program's GUI [26]. Generally, random UI Tests facilitate a *higher Code Coverage and fault detection* in comparison to the other two approaches. Furthermore, it can be implemented very easily, because there is *no manual setup* required[7].

Nevertheless, as testing is completely random, there is a high probability of testing the *same scenarios multiple times*. Additionally, there is *no way to save and replay* the sequence of events triggering a bug[8].

Model-Based UI Testing The foundation of Model-Based UI Testing is the creation of the best possible *GUI Model*[9], representing the visual structure of a program [26]. The GUI Model is provided as a test input, allowing to generate possible UI interaction event sequences.

However, as the need to manually define the GUI Model exists, Model-Based UI Testing could be described as a *semi-automatic* way of testing.

[7] Random tests are fully automatic, described in Section 3.5

[8] Saving & replaying required for automation described in Section 3.5

[9] GUI Model explained in Section 3.1.1

Model Learning UI Testing Last but not least, learning model systems provide an even more sophisticated testing approach, where the *GUI Model generation is completely automatic* [26]. For this reason, there is no need to manually create and update the underlying GUI Model[10].

Nonetheless, Model Learning UI Testing is the most expensive way of software testing concerning the *necessary processing power and computing time*. Furthermore, *event loops* have to be avoided, which would lead to revisiting and testing pages redundantly.

GUI Models[11] are created by *traversing a program's GUI*, thus representing a "correct" definition of the GUI [26]. This *model generation* process includes the following *two questions* described in List 3.1.2.

1. Does the GUI Model converge to the actual GUI?[12]
2. How to decide if two GUI states are equivalent?[13]

However, there is always the possibility of overfitting when checking for GUI state equivalency [26].

Furthermore, such GUI Models can be *generated automatically*, thus providing a way to generate UI Test cases [24, p. 1479]. This is why there are tools available, which can extract, generate and run UI Tests automatically[14].

[10] GUI Models must usually be updated
[11] GUI Model explained in Section 3.1.1
[12] GUI Model must converge as fast as possible
[13] State equivalency indicates correct GUI representation
[14] UI Test Automation described in Section 3.5

3.1.3. UI Testing Layers

Generally, UI Tests can be subdivided into *three distinctive layers*, which are presented within the following paragraphs [20, pp. 55-57]. However, the main purpose of this section is to provide only a rough overview of this subdivision of testing layers.

External Layer The external layer is also called the *User Interaction Layer*, because it *represents the GUI* of a program [20, p. 56]. UI Tests are performed on top of it, because the overall goal is to test from the user's perspective[15].

Behavioral Layer The purpose of this layer is to summarize all *behavioral data* of a program[16], which can be mined and analyzed in order to infer high-level code abstractions called *Invariants*[17] [20, p. 56].

Code Layer The code layer is the *lowest UI Testing level*, representing the layer where the source Code Coverage gets measured[18] [20, p. 56]. For this reason, the code layer is mainly used for quality assurance purposes.

However, not only the external but all of these three layers are involved in the process of *automated UI Testing*[19]. Nevertheless, these levels are also quite sensitive to certain influencing aspects[20], which have to be considered.

[15] UI Testing basics described in Section 3.1
[16] Behavioral data may include runtime properties or function return values
[17] Invariant described in Section 3.1.1
[18] Code Coverage described in Section 3.1.1
[19] UI Test Automation described in Section 3.5
[20] UI Testing aspects described in Section 3.3

3.2. Why Testing the UI?

Nowadays, smartphones and tablets dominate our everyday communications [26]. This is one of the main reasons, why the *demand for thorough testing* has increased significantly, especially regarding the GUI[21]. Due to the *intense competition* today, software providers have to guarantee an application meets its user's expectations[22]. Nonetheless, many applications nowadays are buggy or slow, which is why they usually cannot compete with others, which are able to satisfy their user's needs.

Although many mobile apps today look simple on the surface, most are still *complicated underneath* [21, p. XI]. Therefore, there are generally a lot of possibilities to introduce potential errors. Additionally, mobile devices as well as installed mobile apps tend to be *updated frequently*, which is yet another indicator for the importance of UI Testing [22, p. 46].

This is why engineers *demand reusable and cost-effective testing environments*, which is one of the main reasons to automate UI Testing[23] [20, p. 55]. Nevertheless, these test environments must provide a way to *reliably repeat UI Tests*, producing the same test results unless either the program or the according tests change[24].

Because of these reasons an application has to be tested as a whole, thus *verifying its GUI* is functioning as specified and expected [27, p. 190]. This means that the functionality of all important UI controls should be checked, while not forgetting about the big picture.

As already mentioned in Section 3.1, UI Testing is not only commonly used for System Testing[25], but also for Acceptance Testing[26], because it provides a great way to demonstrate that a certain feature works how it is supposed to work [21, p. 11].

[21] GUI described in Section 3.1.1
[22] Economical aspects described in Section 2.3.2
[23] UI Test Automation described in Section 3.5
[24] Test result consisting of Code Coverage, Invariants & states
[25] System Testing described in Section 2.5.3
[26] Acceptance Testing described in Section 2.5.3

3.3. Important Aspects of UI Testing

One of the most important aspects of UI Testing is the *economical* consideration, because a setup of a sophisticated UI Test framework involves high complexity, which is why it is very expensive[27] [22, p. 54]. Especially today's common practice of *frequently upgrading devices and applications* is very tedious and time-consuming, thus making testing even more challenging.

The goal of any professional software test suite must be to deterministically discover bugs in the most reliable fashion [20, pp. 55-57]. However, all the *various devices and platforms* available today make it very hard to achieve this overall testing goal.

This is the reason why different test executions using exactly the same program may lead to varying Code Coverage[28] or Invariants[29]. Therefore, the most *critical factors* must be carefully controlled to ensure a proper and valuable way of UI Testing. If these aspects are not thoroughly observed, it will lead to test Flakiness[30], not only affecting the test results, but also the experimentation with new UI Testing techniques [24, p. 1479].

Generally, the largest influencing factors can be observed at the *top layer*[31] as well as at the *bottom layer*[32], representing the GUI and the code state of an application [20, p. 64].

According to Gao, Liang, Cohen, *et al.*, there are *four major groups* of important aspects related to UI Testing [20, p. 57]. These critical factors are summarized within the following paragraphs, illustrating what has to be considered when testing an UI.

[27] Economical aspects described in Section 2.3.2
[28] Code Coverage described in Section 3.1.1
[29] Invariant described in Section 3.1.1
[30] Flakiness described in Section 3.1.1
[31] External layer described in Section 3.1.3
[32] Code layer described in Section 3.1.3

Execution Platform The platform where the UI Tests are performed on marks a crucial testing factor, because discrete *operating systems* potentially handle certain aspects differently[33] [20, pp. 55-57]. This does not only relate to completely different systems, but also to *varying versions of the same* operating system.

App & Language Version Similar to the influencing aspect of the system version[34], the current *app version* indicates another UI Testing aspect, which potentially causes test Flakiness[35] [20, pp. 55-57].

However, a varying app version may simply handle events differently or implement other thread policies, which could be due to code changes or even due to the usage of another version of a *programming language*.

Initial State & Configuration Another impacting aspect for UI Testing is the fact that *preferences* may influence the startup of a program[36] [20, p. 57]. Furthermore, there is a possibility that currently running test cases adapt existing settings, causing a *modified setup* for tests still to be executed.

Because of this, potentially influencing configurations must be *identified and restored* before every single test case execution, thus providing a reliable testing environment.

"Harness Factors" Harness Factors are related to automated UI Testing using *automation tools*[37] [20, p. 57]. However, these factors describe *preferences* related to the automation tool[38], not to the tested program. According values may be set by default or heuristically specified by a test engineer.

Nonetheless, these factors are the reason, why there may occur *too long delays* slowing down testing, or *too short delays* not allowing to finish certain functionalities. Especially due to those delays, aspects like a program's memory or CPU power may impact the reliability of a UI Test.

---

[33] Systems may differ in rendering UIs, load times. . .
[34] System's version influence described in Section 3.3
[35] Flakiness described in Section 3.1.1
[36] Preferences like config files, system settings. . .
[37] UI Automation tools described in Section 3.5.2
[38] Automation params like startup/step delays. . .

There are even *more aspects*, which may have an *impact on the reliability* of UI Testing [24, p. 1479]. This is why even changes which seem to be only of low relevance[39], may influence the GUI or Code Coverage, thus causing Flakiness of a GUI-centric application.

As already mentioned, multi-threading may also change a system's state, while unpredictably delaying a task's completion.

However, the following list illustrates some additional *external dependencies*, which have not been covered so far [25, pp. 204-205], [21, p. XI]. This diversity of influencing aspects clearly indicates why UI Testing is such a highly complex process.

- System speed
- Storage/sensor access
- Networking
- Animations
- Pup-up windows
- Orientation changes
- ...

According to Gao, Liang, Cohen, *et al.*, it may just not be possible to obtain multiple test runs, while guaranteeing a constant Code Coverage [20, pp. 55-57]. The reason for this hypothesis is that there are *too many uncontrollable factors*[40]. This is why test engineers cannot simply eliminate all potential output variations. Nevertheless, the overall UI Testing goal must be to reduce these influences to a minimum.

In addition to all the aspects mentioned so far, also common considerations regarding whether to make use of Black-Box or White-Box Testing exist [23, p. 12]. Generally, *Black-Box Testing* indicates what can be tested[41], while *White-Box Testing* shows what should be tested[42].

Usually, it is the best to make use of a combination of both approaches, because Black-Box Testing may lead to irrelevant tests, while White-Box Testing on the other hand may lead to non-executable ones.

[39] Minor changes due to the current time, system load...
[40] Uncontrollable, because too specific or sensitive to minor changes
[41] Black-Box Testing described in Section 2.5.1
[42] White-Box Testing described in Section 2.5.2

3.4. UI Testing Principles

In addition to all the aspects mentioned in Section Important Aspects of UI Testing, there are *practices leading to more consistency*, thus reducing the Flakiness of tests[43] [20, p. 55]. As already explained, it must be a test engineer's main desire to advocate more reliable UI Testing, thus enabling tests to be *repeatable and automatable*.

According to Arlt and Gao, Liang, Cohen, *et al.*, there are *four main principles* that should be followed in order to enable more stable and goal-oriented UI Testing [20, p. 64], [23, p. 11]. These main guidelines are shortly summarized within the next few paragraphs.

Share Information As there is no realistic way to eliminate all possible test variations[44], important *configurations and platform states*[45] must be *reported and shared* in combination with the test outcomes [20, p. 64].

Multiple Test Runs UI Tests must be executed multiple times before expecting them to deliver reliable results [20, p. 64]. Additionally, *averages and variances* have to be reported while explaining how the *test environment* may influence on UI Tests.

App Domain Information Providing app specific *domain information* helps to reduce Flakiness as well[44] [20, p. 64]. Therefore, it is necessary to identify all influencing factors in order to filter them when evaluating the test results.

Executability & Relevance Test engineers need to identify event sequences that are *executable and relevant* at the same time[46] [23, p. 11]. Nevertheless, sophisticated UI Testing must be concentrated on executable and relevant user interactions, while covering as much code as possible [23, p. 33].

[43] Flakiness described in Section 3.1.1
[44] UI Testing aspects described in Section 3.3
[45] Infos like system or language version, starting state. . .
[46] Irrelevant sequences like Paste & Copy (instead of Copy & Paste)

3.5. UI Test Automation

The purpose of this section is not only to explain the basics of the automation of UI Testing, but also to describe all necessary aspects related to such automation tools. Nevertheless, as this thesis deals with automated UI Testing in iOS, the main focus naturally lies on the test automation of *mobile applications*, while focussing on iOS apps.

3.5.1. About Automated UI Testing

"Testing software involves a lot of repetitive work, and repetitive work is excellent work for a computer to do." [21, p. 1]. This quotation by Penn perfectly illustrates the fundamentals of test automation in general, as well as regarding the automation of UI Tests. Basically, test automation aims at making tests *automatically repeatable*, without the necessity of additional human interactions.

However, a study from 2012 predicts an increase in overall mobile testing tool revenues from $200 million in 2012 up to $800 million by the end of 2017 [28]. Especially the *growing demand for automation* of software testing seems to be the predominant reason for this significant raise. Additionally, the importance of making excellent first impressions in today's highly competitive markets makes rigorous *automated testing indispensable*[47].

As already mentioned in Section Why Testing the UI?, most test engineers today have to consider frequent *device, system and technology upgrades* [22, p. 46]. This is yet another reason for the continuously growing need of *reusable and cost-effective* ways of UI Testing.

Furthermore, all the various development and maintenance tasks cause a high demand for *reliable and repeatable* UI Testing [20, p. 55]. Therefore, *specialized tools* are required in order to provide a decent way to generate, execute and repeat test cases [24, p. 1479].

[47] Intense competition described in Section 3.2

UI Test Automation describes a *guided application traversal* by automatically interacting with an application's GUI[48] [26]. According to this definition, app *runtime properties* get recorded while internal app states and test outcomes are *systematically analyzed* [25, pp. 204-206].

Nevertheless, the automation of UI Testing should not be considered as a total replacement for human testers, but it should rather be perceived as a supporting testing tool [21, pp. IX-X]. Generally, the idea is to *automate mundane tasks*, thus allowing test engineers to focus on other, intellectually challenging aspects[49]. Furthermore, it is economically not feasible to automate every single testing procedure which is why *human testers must always be involved* when testing software[50].

Still, even people with only very *basic software development skills* have the possibility to create or edit UI Tests if a state-of-the-art test automation tool is used[51] [27, p. 190]. Furthermore, automated UI Testing can be performed in *various ways*, ranging from running it on an emulator or simulator, to testing on real world gadgets like smartphones or tablets [26].

[48] GUI described in Section 3.1.1
[49] Psychological aspects described in Section 2.3.1
[50] Economical aspects described in Section 2.3.2
[51] Automation tools described in Section 3.5.2

3.5.2. Automation Tools

Nowadays, there are a *multitude* of UI Automation tools available, which aim at supporting test engineers to automate certain UI Testing procedures [20, p. 55]. Nevertheless, many of these tools are *platform dependent*, which is why they cannot be used for cross-platform UI Testing [24, p. 1479].

As already explained in Section About Automated UI Testing, a UI Test Automation tool runs either on a real device, an emulator or simulator, while extracting an *application's UI structure* [25, pp. 205-206]. However, in certain cases the tested app must be instrumented to enable test automation while other tools simply work with the unmodified version of an application[52].

Nonetheless, some tools run *completely automatically* – with no need of any human interactions – while others still require a significant amount of manual input [23, p. 11]. As a matter of course, it is the overall testing goal to generate test cases in a completely automatic fashion.

Usually, *"semi-automatic"* tools facilitate the manual recording of UI Tests with the support of *Capture-Replay tools*. In case of using such a tool, testers have to perform manual interactions on the GUI[53] which are simultaneously mapped into an according UI Test case[54]. After this *test recording*, previously captured test cases can be replayed any number of times, thus enabling an automatic requirement verification.

Despite the possibility to rearrange or edit these generated UI event sequences, their maintenance is still a *tedious and time-consuming* task [23, p. 11]. The main reason for this is that captured UI Tests are generally very *vulnerable to code changes*, due to a tight coupling between the UI Tests and the application's GUI [21, p. 41].

Nevertheless, although the automation of software testing is steadily becoming more important, there is still a *lack of standardization* concerning test infrastructures, scripting languages and app interfaces for testing [22, p. 54].

[52] Instrumentation requires code changes
[53] GUI described in Section 3.1.1
[54] Event sequence usually defined in requirements

Tool Requirements

The following paragraphs illustrate additional tool requirements considering the automation of UI Testing. Nevertheless, as the automation of UI Testing is tightly coupled to the project's environment[55], there is *no all-round solution* available yet which is fitting every project's needs.

Property Support As the process of UI Test Automation is build up on the analysis of runtime attributes[56], an according automation tool needs to support various *high-level UI properties* [25, p. 206]. This is why a predefined set of attributes must be available, abstracting an applications's GUI.

App State Access In addition to having access to UI properties, an automation tool must also allow to access other *arbitrary app states* [25, p. 206]. Accordingly, additionally automation tools must not only provide access to certain *system states*[57], but also to *internal app states* and preferences.

Exploration Flexibility An automation tool has to provide a way to *customize* the UI exploration, providing high flexibility in testing [25, p. 206]. Therefore, it has to support *editable event sequences*, allowing to decide which steps should be initiated next and how to verify a certain UI state.

Trigger Actions In order to guarantee a reliable and stable UI Automation, the app's *robustness to environmental changes* must be verified[58] [25, pp. 206-207]. Another important aspect is that an automation tool must additionally allow the injection of *exceptional inputs*[59] [26].

Scripting Language The *readability and maintainability* of UI Tests indicates another important aspect, because understandable tests allow us to verify a whole system [21, p. 41]. Anyway, if a tool requires the know-how of another programming language, it might be difficult to motivate people to test.

[55]UI Testing aspects described in Section 3.3
[56]UI Test Automation described in Section 3.5.1
[57]System states like network connection, CPU usage...
[58]Environmental changes like change of network connection...
[59]Exceptional inputs like login/registration texts, push notifications...

As already mentioned in Section Automation Tools, the overall automation goal is to generate test cases fully automatically [23, p. 11]. However, such a fully *autonomous testing solution is difficult to implement*, because all generated event sequences must be both executable and relevant at the same time[60].

Although all of the mentioned requirements summarize only the main aspects of UI Test Automation, still many *open research questions and challenges* exist [25, pp. 206-207].

[60]UI Test principles described in Section 3.4

3.5.3. Supporting Patterns

Nowadays, professional test engineers and software developers usually adopt common *programming patterns* in order to enable an application for test automation[61] [13, p. 47]. Therefore, this section's purpose is to shortly describe some of these core techniques of software testing, thus providing a way to *break app dependencies*.

First of all, it is important to describe the common *term 'dependency'*, as it marks the foundation of all the following pattern definitions [13, pp. 49-50]. Basically, if a program interacts with an *object – not directly controlled* by the program itself – this object is called an external app dependency or simply a dependency[62].

Dependency Injection

Usually, *fake implementations are injected* into an app in order to instrument it for testing[63], which is commonly known as Dependency Injection [13, p. 57]. According to Osherove, there are *three approved ways* to inject dependencies, which are shortly summarized in the following List 3.5.3.

1. *Constructor* Injection
2. *Property* Injection
3. *Method* Injection

Nevertheless, it is important to know that there are *several ways of injecting* such external dependencies [13, pp. 57-74]. Still, regardless of which form of Dependency Injection is used[64], the main aspect is that external dependencies have to be split up in order to *enable automated testing*.

Furthermore, a Dependency Injection additionally describes a commonly accepted pattern to *decouple software components* in general, even when not focussing on specific test automation needs [13, pp. 57-74].

[61] Pattern is a reusable solution for a common development problem
[62] Dependencies like a web service, threading, memory access. . .
[63] Doubles explained in Section 3.5.3
[64] Forms of Dependency Injection with different pros & cons

Doubles

Generally, Doubles represent *simplified code replacements*, which are usually a requirement for test automation[65] [29, p. 1]. Their main purpose is substituting unready components or isolating certain components from other app parts or from the test code. Additionally, Doubles provide an interface to *control side effects* like for instance return values, while some may even allow to *verify requirements*.

Nonetheless, the generic term 'Doubles' can be subdivided into *two more specific terms*, which are explained in the following paragraphs [13, p. 78].

Stubs A major issue of software testing is that test cases have *no control over external dependencies* including what to return or how to behave[66] [13, pp. 49-50], [29, p. 1]. Therefore, Stubs represent a *simplified and controllable replacement* for such dependencies, simulating according behavior by returning the same outputs as the imitated components. According to this, Stubs are only *static replacements*[67], being very closely bound to the related test cases. Therefore, this pattern is *limited in functionality*, because Stubs often have to be modified if a test case changes.

Mocks Generally, the purpose of Mocks is to be a *simplified and controllable replacement for objects*[68]. This typically is useful when a complex object cannot be easily considered for testing[69] [13, pp. 75-77], [29, p. 1].
Although this definition seems to be similar to the one of a Stub[70], Mocks can additionally decide whether a test succeeds or fails by *asserting against specified requirements*[71]. Therefore, even if an object does not return or save anything, it can still be verified, due to providing an *external Application Programming Interface (API)* for testing purposes.

[65] Doubles also known as Fakes
[66] Dependencies described in Section 3.5.3
[67] Stubs not interacting with other components
[68] Mocks limited to object-oriented programming
[69] Might be impractical or impossible to include an object
[70] Stubs described in Section 3.5.3
[71] Mocks assert against called methods or properties

Nevertheless, it is very important to *not over-use Mocks*, in order to prevent a potentially significant increase in maintenance needs [29, p. 1]. Furthermore, Mocks must be *as accurate as possible* to produce no misleading results, which can be very difficult to achieve[72].

Professional software engineers should be aware of how to differentiate Stubs from Mocks [13, p. 78]. However, as these terms are often used interchangeably by mistake, many people tend to be confused, while remaining unaware of these *noticeable distinctions*. The main difference is that a Mock can actively fail tests, which cannot be done by a Stub.

[72]Mocked object may be from different developer/project, not even existing...

Part II.

Technical Realization

4. Project Introduction

In order to be able to understand the general motivation behind most of the specific UI Automation decisions made in course of the technical realization, it is important to have at least a basic knowledge of the most important aspects and considerations. Therefore, it is this section's main purpose to provide the necessary overview, while focussing on the things most relevant for the automation of UI Testing regarding the iOS platform.

4.1. Project Architecture

The project has been developed over several years, using an *agile development* approach – where the software was developed in short cycles – with a mixture of *two different development languages*[1] [7, pp. 175-176]. This simultaneous development with two languages is the main reason why it could not only be described as well-established, but also as very complex[2].

As this thesis is mainly concerned with the UI Automation process of iOS applications, the according project is targeting devices supporting a version *greater than iOS 7*, which is Apple's mobile system version that was released in the end of 2013 [30]. Here, not only iPhones and iPads are targeted, but also other platforms like *Apple TV*[3] *or OS X*[4], which is yet another reason for the project's *high complexity*.

Nevertheless, this thesis' focus mainly lies on mobile *iOS applications* developed for iPhones as well as for iPads. However, all decisions concerning the automation of testing where still made while keeping all of these differentiating platforms in mind.

[1] Project developed using Objective-C & Swift

[2] Mixture of two languages requires bridging & trade-offs

[3] `http://www.apple.com/tv`

[4] `http://www.apple.com/osx/what-is`

4.2. Testing Process

Until the start of writing this thesis, one of the overall goals was to guarantee a high product quality through *manual code reviews* done by other software engineers. Additionally, the application was *internally and externally tested* by developers as well as by specialized software testers. So, all testing was done completely manually, which is why there was *no automated testing* process involved.

Every feature was manually tested during the product development and usually right after the completion of a certain feature. Furthermore, the whole application was regularly checked by several team members shortly before every product release or update.

All necessary *test requirements* had been documented by the responsible test engineers, including a detailed summary of all the necessary testing steps as well as of the required guidelines for testing[5]. These detailed test specifications were *copied and saved* for each of the release candidates, allowing to manually reproduce previously developed test scenarios.

4.3. Room for Improvement

Obviously, there was enough room for improvement, not only regarding the whole app structure, but especially concerning the overall testing process. This section summarizes all the major considerations related to the project's *quality assurance and enhancement*.

However, as already described in Section Psychological Aspects, one of the main issues of regular manual software testing is, that humans make more errors if they have to do exactly the same repetitive tasks over and over again[6]. Therefore, *relying solely on manual testing* is generally not recommended, but it should rather be used in combination with a sophisticated automated testing solution[7].

[5]Test guidelines like the specific device or iOS version...
[6]Iterative manual testing is commonly known as regression testing
[7]UI Test Automation described in Section 3.5.1

In addition to the *high error-proneness*, the totally manual software testing approach was also very *time-consuming* and did cost a lot of money.

As already mentioned in Section About Automated UI Testing, all repetitive and redundant tasks should be *ideally done by test automation tools* rather than by human testers [21]. The main reason for this is that it saves a tremendous amount of time, while one is able to focus on other relevant and more challenging tasks of testing.

Furthermore, the project's code base was *not optimized for automated testing* purposes at this time. Especially, the various deeply integrated *external dependencies* represented one of the most critical aspects regarding the introduction of a sophisticated test automation process[8]. Additionally, executing the program involved many *asynchronous tasks*[9], which also strongly influenced the overall testing routine[10].

Another issue was related to data tracking, because all the manual testing was done by using applications installed on real devices with *real API connections*. Because of this, manual test actions were *tracked like real user interactions* and thus falsifying the according tracking data.

Furthermore, as the manual testing results were depending on the live API connection, some potential bugs could only be narrowed down to the server-side, which was not developed in-house. On the one hand this is good, because the product gets tested as a whole – including the API and other external dependencies – but on the other hand, potential app bugs were debugged by the app development team. From an economical point of view this was a critical issue, because a *lot of time for searching bugs* was required, while the bugs *could not be found* after all[11].

[8]Dependencies described in Section 3.5.3
[9]Asynchronous tasks like threading, file/database access. . .
[10]UI Testing aspects described in Section 3.3
[11]Economical aspects described in Section 2.3.2

4.4. UI Testing Ambitions

According to the main weaknesses of the project's setup and testing process illustrated in Section Room for Improvement, this section aims at explaining all the derived testing goals referring to these weak spots.

The main goal was to introduce an *UI Test Automation* solution[12] which could be implemented using both, the latest *Swift and Objective-C* development languages[13]. The reason for this decision was that many core parts of the application were still relying on the former Objective-C language, which is why it had to be supported in combination with Swift.

The overall ambition was to *hand-off redundant and repetitive testing tasks* to this optimized UI Test Automation approach, thus allowing the test engineers to focus on exploratory and more challenging testing.

The basic idea was to *verify every single code change* before and iteratively after it was integrated into the main application code base[14]. For this reason, software engineers should only be allowed to integrate any of their implementations if every aspect had been tested accordingly, without breaking any already existing test cases.

Nonetheless, the overall goal was not only to enable UI Automation, but also to set it up to be as *fast and stable* as possible in order to guarantee most reliable test results. Additionally, the UI Test Automation suite had to be very *understandable and maintainable* to enable a long-term utilization. Furthermore, the *reusability* of the automation framework marked another key factor, because the core testing structures could potentially be established within other or future iOS projects as well.

Also, in order to guarantee a high quality of the integrated UI Automation framework – and of the existing product – the idea was to introduce an *automatic code style verification* process in addition to the manual code reviews[14]. Therefore, it should not be possible to integrate any product or test code changes, which do not conform the predefined *code style guidelines*.

[12] UI Test Automation described in Section 3.5.1
[13] Objective-C is the precursor language of Swift
[14] Automatic verification with Continuous Integration (CI) mechanisms

5. UI Automation in iOS

First of all, this chapter provides a rough overview of the main automation tools available, which have been considered for automating the project's individual UI Testing process. Additionally, the key differences between these tools are illustrated[1], while explaining the reasons for the selection of the automation framework which has been embedded and optimized afterwards. Furthermore, some tool-specific details are provided concerning the UI Testing process, its remaining issues and potential optimization steps.

5.1. UI Automation Frameworks

Generally, *various possibilities* to tackle the problem of automating the UI Testing process in iOS exist [31]. Nevertheless, most of the available solutions aim at interacting with an application as a real user would do. For any third-party automation tools, this is only possible through leveraging the *iOS accessibility infrastructure*[2], allowing tools to interact with an application through a special "VoiceOver" interface [32].

After an automation tool is successfully integrated within the Apple's software development environment called *Xcode*, the testing procedure usually launches the application, while the automation tool attaches to the executing app's process [21, p. 2].

Additionally, automation frameworks can usually be easily integrated into *Continuous Integration (CI)* systems, which provide the support for a completely automatic UI Testing approach, without any need of human interactions [13, pp. 126-129]. Generally, CI describes an automatically build and integration process running continuously on a specialized CI server.

[1] UI Automation tools described in Section 3.5.2
[2] http://www.apple.com/accessibility/ios

Nonetheless, after doing an extensive research into the field of UI Automation, only very *few automation tools* turned out to be commonly accepted and supported by a significant community[3]. However, these popular tools could be *narrowed down* to a subset of two, which are explained and compared in more detail within the following sections.

5.1.1. KIF - Keep It Functional

KIF describes a popular *open-source* iOS Integration and Acceptance Testing framework, which was created by Square in 2011 [33], [34]. It supports *Objective-C and Swift* with a minimum version of *iOS 5* which is reducing the overall learning and adaption curve of software testing.

Nonetheless, as KIF makes use of Xcode's *default XCTest target*, all built-in test functionality is available for UI Test Automation purposes[4] [34]. Therefore, the setup of the automation process requires only *minimal effort*[5], without the need to install additional dependencies to begin testing.

KIF tests run *synchronously on the main thread*, while constantly looping through the available view hierarchy to explore the GUI[6] [34].

Nevertheless, this third-party framework is dependent on *undocumented Apple APIs*, which are generally considered to be safe for testing but not for production usage. This is why it is important to not include test code within any final app store builds, because otherwise Apple will deny the app's public submission.

Additionally, the KIF structure can be *easily extended* with custom functionality and integrated into *CI systems*[7], which enables all UI Tests to be executed completely autonomically [34]. Especially, the fact that KIF is considered to be very extensible is the main reason for its popularity and *well-established open-source community*.

[3] Automation tools like UI Testing, KIF, Appium...

[4] Xcode build tools, test navigator & reports...

[5] KIF setup includes integration and basic Xcode configuration

[6] GUI described in Section 3.1.1

[7] CI shortly described in Section 5.1

5.1.2. UI Testing

UI Testing is the name of Apple's *official UI Automation tool* which was introduced with *Xcode 7* in September 2015 [35], [36]. Nonetheless, the UI Testing framework is also concerned with Integration and Acceptance Testing while targeting any mobile applications supporting *iOS 9 or later* and even the OS X desktop system with a minimum version of *OS X 10.11*.

Additionally, the framework's main APIs have Swift interfaces that can be used within projects supporting both *Swift and Objective-C* [31], [32]. Nevertheless, the according XCUI namespace just serves as a *minimalistic and limited UI Testing interface*, which strictly separates the testing target from the tested mobile application. This is the reason why test cases are only able to interact with an application via *three special proxy elements* provided by the automation framework[8].

Last but not least, UI Testing provides a *test recording functionality*[9], which is able to capture manual UI interactions after pressing the according record button in Xcode [32]. Therefore, the process of the UI Test *creation is much easier and faster* in comparison to any third-party automation tools like KIF[10], where all tests have to be created programmatically. Furthermore, the possibility to record UI Tests enables people with limited development skills to create UI Tests themselves.

5.1.3. KIF VS UI Testing

Table 5.1.3 clearly summarizes all the *distinctive properties* of KIF in comparison with Apple's default UI Testing framework. This simplified overview also served as the foundation of the decision making process, determining which automation framework to use for the specific project[11].

Therefore, this section's purpose is to highlight the main aspects which lead to the final *automation tool selection*. It is very important to consider the individual *project context* when analyzing the final decision.

---

[8] XCUIApplication, XCUIElementQuery & XCUIElement
[9] Capture-Replay tools described in Section 3.5.2
[10] KIF framework described in Section 5.1.1
[11] Project described in Section 4.1

	KIF	UI Testing
release	*2011*	2015
offer	third-party	*official*
source	*open-source*	closed-source
targets	iOS	*iOS & OS X*
min version	*iOS 5*	iOS 9 & *OS X 10.11*
setup	minimal	*integrated*
APIs	undocumented	*official*
API limits	*VoiceOver*	XCUI proxies
extensibility	*unrestricted*	limited
Capture-Replay	no recording	*recording*
community	*established & large*	young & growing

Table 5.1.: iOS UI Automation: KIF VS "UI Testing"
This table illustrates the main differences between the two *iOS UI Automation tools KIF and UI Testing*. Similar properties are omitted and special traits are highlighted in order to focus on the most *relevant aspects*.

Generally, it was a relatively straightforward decision-making process to select *KIF* as the more appropriate solution for the specific project[12].

However, the most important consideration was that only KIF allows to automate UI Testing for iOS devices using *iOS 7 or higher*, which was one of the project's main predefined testing requirements.
Furthermore, KIF tests are executed attached to the *application process*, thus allowing to *extend and optimize* the existing framework functionality.
Finally, the *large open-source community* marked another important factor as well as the aspect that UI Testing was still relatively young in comparison to KIF, which is why KIF was generally *better documented and supported*.

Nevertheless, UI Testing might still be well-suited for UI Testing of other iOS or OS X applications, which do not require a *backward compatibility* below a version of iOS 9. Especially, the aspect that UI Testing is the officially integrated automation tool – providing additional features like UI Test recording – indicates its significant UI Automation potential.

[12]Project described in Section 4.1

5.2. KIF Integration

This section summarizes all the *main findings*, which have been gathered within the scope of the *integration and optimization* of the well-established UI Automation framework called KIF[13].

First of all, the basics of KIF's UI Automation process are illustrated, including an overview of the major *shortcomings* when testing with this open-source automation framework. Afterwards, potential *modification and optimization steps* are explained, highlighting all inherent considerations.

Although this potential *optimization is tailored* to the actual needs of the previously described project[14], most of the specified enhancements can still be used within different projects as well, depending on the individual testing philosophy of the responsible software testing team[15].

Nevertheless, the *high complexity* of the project marked a *critical influencing factor* for many decisions related to the automation of UI Testing.

However, as the project has been developed over several years, naturally code parts exist that do not conform to the latest *iOS design principles*. Because of this, some *restructuring* was required in order to allow an integration of sophisticated automated testing routines. Nonetheless, this optimization was relatively *time-consuming and challenging* to achieve, because some of these code parts were deeply integrated within the core of the according iOS application.

The combined usage of *Objective-C and Swift* is representing another limiting factor, because not all language constructs are portable between both of these programming languages[16].

Furthermore, the initial *lack of experience* in the fields of test automation and Continuous Integration must be considered as well. However, although this is obviously a limiting factor, it pushed the research team forward and motivated them to consider the *latest tools and optimizations* available.

[13]KIF framework described in Section 5.1.1
[14]Project described in Section 4.1
[15]Trade-offs required to increase performance, stability...
[16]Objective-C does not handle Swift Structs, Tuples, Generics...

5.2.1. Testing with KIF

Generally, the basic setup of the KIF framework requires only a very *minimal effort* until being able to automate UI Tests, because there is no need for any additional dependencies[17]. As already mentioned before, KIF makes use of the default *XCTest target*, thus allowing to use all built-in tools of Xcode[18].

However, the KIF automation framework is *synchronously looping the GUI* hierarchy, providing a way to visualize the current app structure and *making assertions on it*, which is necessary for automated UI Testing [34].

Nonetheless, at least some knowledge of the underlying app structure is required, which is why KIF is commonly known as a *Grey-Box Testing* tool[19], where certain parts of the application have to be considered for testing [37].

KIF tests are running attached to the *application process*, allowing testers to modify the underlying application code, while an additional *compile time code verification* functionality is provided as well [21, p. 196].

However, such *runtime app manipulations* should generally be avoided, because it is a common practice to separate the test code from the application code whenever it is possible. Furthermore, the test code must be excluded from potential app release candidates, because *Apple prohibits* to make use of its *undocumented APIs* within any published iOS applications.

Nevertheless, the overall goal is not only to verify an app's functionality from the perspective of a user, but KIF additionally aims at *enforcing the accessibility* of an application for people with visual disabilities [38].

Thus, the basic UI Automation framework is providing two alternatives for accessibility handling, called *accessibility label and accessibility identifier*. However, only accessibility labels are visible to the users, which may therefore enhance the overall accessibility of an iOS application.

This is why, a special *extension pack* is required in order to activate UI Testing with *accessibility identifiers*. Nevertheless, the main advantage of using accessibility identifiers over the according labels is that there is *no need to translate* them, because they are invisible for real users.

---

[17]KIF framework described in Section 5.1.1

[18]Xcode functionality like build tools, Code Coverage, test reports...

[19]Grey-Box Testing is mixture of Black-Box & White-Box Testing

In addition to the already mentioned aspects, the KIF UI Automation framework includes a special *test setup and teardown* functionality, providing a way to reset the application state for testing. Therefore, it is possible to establish an optimized UI Testing structure, where individual test cases are completely independent from each other, which is one of the overall goals of software testing[20].

Last but not least, as there is *no test recording* functionality available, the whole test suite has to be *manually developed* entirely, which is usually a very *time-consuming* process[21]. Therefore, test engineers should define priorities, describing which app functionality has to be tested, while also specifying on a concrete *test priority* order.

5.2.2. Framework Drawbacks

Generally, it was a very *satisfying decision* to construct the tailored UI Automation process on the foundations of KIF[22], because the default framework already provides a large set of the necessary UI Automation functionality.

Nevertheless, due to the *special requirements* of the project[23], the basic functionality provided by KIF was just not enough.

According to Patton, the fact that there are many *invasive automation tools*, may result in software failures being related to the testing tool itself instead of the software under test [9, p. 239]. This general observation has of course also to be applied to the usage of KIF.

Nevertheless, the relatively *tight coupling* between the testing and the application code marks another critical aspect, because even minor app changes may *require significant changes* of the according UI Tests [21, p. 41]. Obviously, this consideration is not only related to this specific UI Automation framework, but it does still indicate one of the major drawbacks of setting up an automated testing process using KIF.

[20] UI Testing aspects described in Section 3.3
[21] Economical aspects described in Section 2.3.2
[22] KIF framework described in Section 5.1.1
[23] Project described in Section 4.1

Generally, software testers have to expect various *asynchronous operations* when testing comparably complex mobile applications[24].

Nevertheless, as KIF is relying on *hard-coded timeouts* to wait for certain UI elements to appear, it is very prone to non-determinism, which is leading to *unreliable results and Flakiness* [7, p. 214], [39]. The reason for this is that KIF's static default timeouts may lead to unnecessary long waiting – slowing down the overall testing performance – or even to occasionally failing UI Tests caused by timeouts that are too short.

As already explained, KIF is *synchronously looping* through the current GUI hierarchy when executing its according UI Tests[25] [39]. Nonetheless, this illustrates the main reason for the very *slow performance* of KIF, because it takes a tremendous amount of time to constantly update the underlying GUI Model[26], which is required for automated UI Testing.

However, it is a common software testing goal to make individual test cases independent from each other, thus providing more reliable results [34]. Nonetheless, as KIF is executing the test cases in an *alphabetical order*, this independence cannot be easily guaranteed. As there is no obvious answer to the question of why KIF does not randomly execute test cases, this aspect indicates another drawback for its usage in UI Testing.

Another inconvenience of KIF is related to the *accessibility infrastructure* provided by Apple, because it is not easily feasible to specify an accessibility label or identifier for all the various UI elements. Therefore, custom utilities are required to cover most of the app's functionality[27].

Although the KIF framework is supported by a relatively large open-source community, still *crucial bugs* which have not been identified or resolved so far exist[28] [34]. Additionally, there is no functionality available for now, which allows to interact with some of the main UI elements of iOS.

[24] UI Testing aspects described in Section 3.3
[25] GUI described in Section 3.1.1
[26] GUI Model described in Section 3.1.1
[27] Utilities for navigation buttons, toolbar items, alerts...
[28] KIF is error-prone when scrolling, swiping...

Finally, KIF's *integration into Xcode* does not always work reliably. Although the according UI Tests are managed as a built-in *Unit Testing target*[29], certain functionalities are usually not working as expected[30]. Nevertheless, this issue seems to be related to Xcode rather than to KIF, but it still marks another negative aspect for the overall UI Automation.

5.2.3. UI Test Optimization

This section summarizes all the main aspects regarding the tailored optimization of the well-established UI Automation framework called KIF while considering all the specific project requirements[31] [34].

All according explanations aim at illustrating the underlying fundamental considerations and concepts, without going into every single detail. However, a few handpicked and more specific examples are illustrated within the following chapter Implementation Details.

For such a complex and well-established project, it is essential to setup a *high-performance and reliable* testing structure, because tests get usually executed dozens of times [9, pp. 220-239].

Still, test engineers must be aware that software changes frequently, which is why *flexibility and maintainability* are very essential, too[32]. Furthermore, the quality standards and guidelines of software development should also be applied to all tests in order to guarantee an overall high product quality.

Therefore, the overall goal was to construct and establish a *reusable testing toolbox* by hiding complex implementation details behind understandable and *simple testing interfaces* [21, pp. 41-58]. This additional abstraction layer was also intended to enable test engineers to think about the system as a whole instead of focussing on any minor details.

"Growing a test suite is just like growing production code. Small steps are better. Build often. Refactor and reorganize for clarity." [21, p. 58]. This statement clearly explains the applied automation optimization approach.

[29] Unit Testing described in Section 2.5.3
[30] Xcode tools like test runner, test reports...
[31] Project described in Section 4.1
[32] UI Testing aspects described in Section 3.3

Performance & Reliability

The purpose of this section is to illustrate the main *extensions of KIF* regarding its performance and reliability. However, every aspect is only explained briefly, without presenting any specific implementation details.

First of all, it is very important to implement a *GUI caching functionality* to always remember the latest state of the underlying GUI Model[33]. Therefore, the crucial performance deficiency of constantly looping through the current GUI hierarchy can be reduced significantly [39].

Nevertheless, for some special cases it is essential to provide a possibility to clear this GUI cache in order to enforce a complete reconstruction of the GUI Model. This *cache cleanup functionality* is also crucial in order to reset the app and testing state to improve the overall reliability.

Despite of the fact that the next relevant optimization step is obviously very controversial[34], it still illustrates an extraordinary factor to improve the overall UI Automation performance and reliability [39]. Nonetheless, in order to avoid making use of KIF's default waiting functionality – leading to unreliable and slow UI Tests[35] – the default notification handling of iOS can be utilized to *broadcast special testing events*[36].

Basically, the idea is to immediately inform a running test case about potentially *relevant state changes*, which often relate to asynchronous app operations. For this reason, such test cases only have to *wait exactly as long as necessary*. Additionally, this concept allows to *trigger app actions*, which is simplifying the commonly desired app cleanup and reset.

Finally, the internally specified *broadcast guideline* was to send such notifications as often as possible, but only if minor code changes were required and thus not manipulating any application logic.

Although, there is already an *alternative available*[37] – providing a very similar waiting interface – it turned out to be slower compared to the custom-built implementation working based on notifications.

[33]GUI Model described in Section 3.1.1
[34]Contradicts the principle of separating app & tests
[35]KIF drawbacks described in Section 5.2.2
[36]Notify changes of visibility, animations, scrolling, data retrieval. . .
[37]Default functionality called "waitForExpectationsWithTimeout"

Generally, complex applications base on *external dependencies*, which cannot be inspected or controlled[38] [21, pp. 147-151]. This is why, automated tests may unexpectedly fail, although the observed app behavior is functioning as expected. However, as it is usually not desirable to verify the accuracy of these external services as well, there has to be an effortless way to fake all the related service responses[39].

According to this explanation, the predefined goal was to provide a simple way to *fake networking responses*. Nonetheless, as different possibilities to decouple an application from its external server exist[40], the individual project architecture has to be carefully taken into account.

However, in the case of the specific project[41], a minimalistic HTTP server was used in order to *intercept real network communications*, while returning static responses for testing purposes. Nevertheless, as such fake responses cannot be easily differentiated from real API responses, an app usually behaves exactly the same, which is required for a reliable UI Automation.

Last but not least, the iOS system allows KIF to make use of special process *environment variables*, providing an additional way to *trigger app changes* [21, p. 135]. Therefore, an according test target can specify custom environment values[42], notifying the app to perform certain actions.

However, one of the various cases for this app environment modification is to *increase the animation speed* of an application which does significantly reduce the overall UI Testing time-consumption[43] [40].

Although it is also possible to disable animations altogether, it is generally not recommended to do so. The main reason for this is that according transitions are most often necessary in order to prevent an application from misbehaving due to the replacement of asynchronous animations with completely synchronous code handling.

Furthermore, commonly many bugs are triggered by animations which is why they should be involved within a reliable UI Automation process.

[38] Dependencies described in Section 3.5.3
[39] Supporting patterns described in Section 3.5.3
[40] Intercept calls of networking protocol, fake server...
[41] Project described in Section 4.1
[42] Environment specifying fake server usage, device orientation...
[43] Increase the speed of the system core animation functionality

Understandability & Maintainability

First of all, the optimized UI Automation framework includes tailored custom *base classes* for testing, providing the necessary *core functionality* used by most of the test cases. Although this aspect is obviously also related to the previous Section Performance & Reliability, it still significantly increases the long-term maintainability of the whole test suite as well.

Especially the custom-built *automatic test setup and teardown* functionality has to be highlighted, because it allows the responsible test engineers to completely focus on UI Testing, without having to be concerned about how to appropriately reset the entire application state[44]. Nevertheless, without providing such an essential core testing functionality, test cases would have to consider unpredictable scenarios like left open alerts[45], modified preferences or other similar threats.

As already mentioned before[46], it was not only a predefined goal to develop a highly maintainable but also a very *reusable testing solution*. Therefore, the encapsulation of all the various testing modules indicates another major influencing factor for many decisions which have been made within the development process of the optimized UI Automation framework.

This is why, it was one of the logically derived requirements to provide very *simple and comprehensible interfaces* for all the UI Testing functionality, including custom base and utility classes as well as extensions[47].

Furthermore, this encapsulation of the implementation details potentially *simplifies a framework exchange*, because in such a scenario solely the centralized delegation code for the underlying UI Automation framework has to be modified, while most of the testing code remains untouched.

Nevertheless, in order to optimize the overall usability of the UI Automation framework, *trade-offs* have to be made between the *simplicity and comprehensibility* of interfaces, because a completely reduced interface might compromise the understandability and vice versa.

[44] UI Testing aspects described in Section 3.3
[45] Open alerts due to previous failures, test error...
[46] UI Testing ambitions described in Section 4.4
[47] Extensions for accessibility, waiting, scrolling...

In addition to the already mentioned ground rules, the entire project team has verbally agreed on certain additional *guidelines*, which are mainly concerning the maintainability and reusability of the UI Automation test suite.

First of all, *accessibility identifiers* should generally be favored over according accessibility labels[48], especially when considering the individual project requirements. As already mentioned, accessibility identifiers must not be localized, which saves time that can instead be used for testing purposes, while *minimizing the localization maintenance efforts*.

However, another rule specifies that it is generally preferred to construct *fake data by using common development patterns*, instead of simply reading in the static data from a file[49]. Despite the fact that it is possible to *record API responses*, the resulting fake data still has to be maintained. This is why, it is usually the best approach to record the data, then translate it into application code, which is generally easier to manage in future.

The aspect that the resulting UI tests are extremely clean and understandable leads to another very important consideration, because there is *no more need to document* any inline code [6]. Furthermore, this form of documentation has the significant advantage that it is executable itself, which is why the UI tests are instantly indicating if the "documentation" has to be updated, thus strongly *increasing the documentation maintainability and trustability*.

Additionally, one guideline describes the need to *precisely document* all the available testing interfaces, thus enabling Xcode to provide *functionality previews and explanations*. However, this is one of the main reasons why the initial time to get familiar with the optimized framework has significantly decreased. Nevertheless, the obvious drawback is related to the overall maintainability, because such code documentations occasionally have to be updated according to certain functionality changes.

Last but not least, the *predefined file structure* indicates another important aspect, because all test files have to fit appropriately into the scheme, which additionally enforces the desirable module encapsulation.

[48] Accessibility handling explained in Section 5.2.1

[49] Best practices for construction called builder or factory pattern

Furthermore, specific *naming conventions* have been introduced to guarantee a high level of maintainability and overall application code quality[50].

Nevertheless, as humans tend to not always follow such informal rules, many of these conventions have also been verified by manual *code reviews* as well as by specialized tools for *static code style analysis* [41].

Additional Enhancements

In addition to all the aspects already mentioned in the sections before, an *enhanced debugging functionality* was also predefined as one of the main goals for testing. However, the best UI Testing suite would be useless, if there was no supportive debugging and logging functionality available.

First of all, a *slow testing mode* got introduced, which is adding a time delay between every single testing step of KIF. The reason for this is, that the optimized UI Tests run exceptionally fast, which is why the human eye cannot follow them anymore. Nevertheless, this mode is more important *for the initial test development* than for the test execution afterwards.

Furthermore, most of the simplified testing interfaces are configured to *delegate important debug information* to the underlying implementation[51], thus enforcing failing test cases to stop where the error actually occurred.

Additionally, it was an overall guideline to keep Xcode's *console output as clean as possible*, thus avoiding the effort to browse through a very cluttered presentation of runtime information when debugging.

As already roughly explained[52], the specific project has been developed using an *agile development* model, which describes one of the most well-known and modern forms of software development lifecycles[53] [7, pp. 175-176].

Generally, such an agile software development approach is based on short *iterative development cycles*, in which all the previously existing functionality has to be verified[54], while new implementations must be tested as well.

[50] Naming conventions for files, variables, interfaces...
[51] Debug information like file name, line number...
[52] Project described in Section 4.1
[53] Development lifecycle models described in Section 2.4.1
[54] Iterative manual testing is commonly known as regression testing

This is the main reason, why a *Continuous Integration* system has been installed[55], providing a simple way to completely automate the execution of UI Tests. However, the remote CI server is configured to check every single code change, where it does not only run all existing test cases but where it also triggers a *static code style analysis* to verify the software quality [41].

For this reason, it was possible to reduce the time frame for each of the individual development cycles, allowing to *release faster and more frequently*, because an overall high product quality could be guaranteed at any time.

5.2.4. Remaining Issues

Although the KIF framework has been enhanced in various ways, still some *optimization potential* exists. Nevertheless, this section summarizes all the major remaining issues, considering the specific project requirements[56].

First of all, even the optimized version of KIF can still not perfectly imitate certain iOS UI interactions [34]. This is why it is occasionally difficult or even *impossible to test specific UI elements*, especially concerning custom views or relatively new iOS widgets[57].

Furthermore, it is often frustrating to be involved with UI Testing in iOS, because Xcode provides a very *unreliable test tool functionality*[58], at least when using the KIF automation framework. For this reason, it is generally really difficult to run all existing UI Tests simultaneously, because Xcode will just not completely index all of them at once. Additionally, the individual test selection is usually very unreliable too, because of the same reason.

As already mentioned before[59], the default KIF framework can already be described as a Grey-Box UI Testing tool, where there is *no strict separation of the application from the test code*. Nevertheless, some of the optimization steps reduced this separation even more, which obviously indicates a very *controversial aspect* of some of the potential framework enhancement.

[55]CI shortly described in Section 5.1
[56]Project described in Section 4.1
[57]KIF issues when scrolling, swiping, using gestures...
[58]Unreliable Xcode tools for building, test running...
[59]Testing with KIF described in Section 5.2.1

Furthermore, as it is usually also necessary to adapt the according application code for UI Testing purposes, the initial *learning curve* to start writing tests has *slightly raised*. However, this is obviously one of the trade-offs to be made in order to test with a *high reliability and tremendous speed*.

Finally, the current optimized UI Testing framework is *not testing against the production API* of the application. Therefore, all involved engineers always have to keep in mind, that according *fake responses may be outdated*. This is why, the according UI Tests might be succeeding, although the already published application is not functioning as expected.

6. Implementation Details

As none of the other chapters is providing any particular examples of how the *optimized KIF* UI Tests may look like in the end, this chapter delivers insight into a few very selective *UI Testing details*.

The goal is not to get lost in any too specific structural elements of the underlying framework. This is why the given examples aim at highlighting only the *most relevant aspects* of UI Testing using the optimized version of KIF, which is discussed in Section UI Test Optimization.

First of all, a major aspect of the provided examples is to emphasize the utilization of the *default notification handling* functionality of iOS for UI Testing purposes[1]. This is why there is no need to wait with any predefined static timeouts, which would lead to flaky and unreliable UI Tests[2]. Therefore, the example test cases are *highly performant and reliable* in comparison to an according implementation using KIF's default testing interfaces.

Furthermore, the following examples demonstrate how clean the according optimized UI Test cases look like, leading to very *understandable and maintainable* UI Tests[3]. Obviously, all framework implementation details are hidden under a very simple interface, wrapping the underlying UI Test framework functionality. Additionally, there is no need for nesting certain UI Testing actions, which increases the readability of these test cases even more.

Last but not least, in addition to the other already mentioned highlights of the UI Test example implementation, a few common *software development patterns* are also provided, including the Factory, Stub and Mock patterns[4].

[1] Performance & reliability optimization described in Section 5.2.3
[2] Flakiness described in Section 3.1.1
[3] Understandability & maintainability optimization described in Section 5.2.3
[4] Stubs & Mocks described in Section 3.5.3

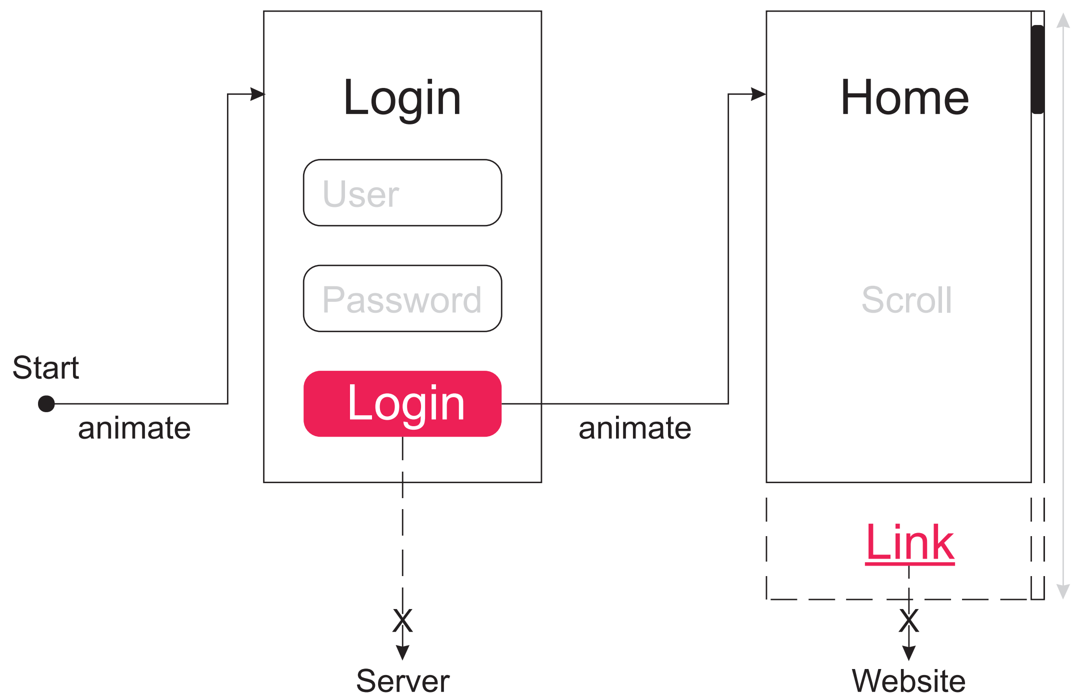

Figure 6.1.: This image illustrates the basic UI of an iOS application, which is used to demonstrate simple UI Testing examples. The iOS app contains two screens, where the first one provides a basic login functionality, which leads to the second screen that is representing the scrollable home interface.

6.1. Text Input & Stubbing

The following List 1 demonstrates an example of how to reliably test the UI of a *successful login flow* of the application illustrated in Figure 6.1.

```
1  func testLoginSuccess() {
2      wait(for: .AnimationEnd, of: "Login")
3      wait(for: .Label, withID: "Login", andText: "Login")
4      wait(for: .Button, withID: "Login", andText: "Login")
5
6      enter("Daniel Fritzsch", into: "User")
7      enter("Master's Thesis", into: "Password")
8
9      let response = ResponseFactory.loginSuccess()
10     stub(method: "POST", path: "/login", response: response)
11
12     perform(tap("Login"), waitFor: "Home")
13 }
```

Listing 1: UI Test Example: Text Input & Stubbing

First of all, the test case waits for the initial *screen animation* to finish (2).

Afterwards, the screen *title label and the login button* are tested, including their visibility and a verification of the displayed texts (3–4).

The next steps are to *insert test inputs* into the two required login text fields that would lead to a failing test if the UI elements to verify would not be available on the screen (6–7).

After the text insertion, the Factory pattern is utilized to construct a *fake login response*, which is used to *stub the login API request* afterwards, preventing the app from addressing the real server (9–10).

Finally, the test is aiming at *tapping the login button*, while it is synchronously *waiting for the home screen* to appear (12).

6.2. Scrolling & Mocking

List 2 provides an example of how to reliably *scroll to and verify* an UI element, which is at first not visible on the screen, as shown in Figure 6.1.

```
1  func testHomeWebLink() {
2      wait(for: .AnimationEnd, of: "Home")
3      wait(for: .Label, withID: "Home", andText: "Home")
4
5      perform(scrollDown("Scroll"),
6              waitFor: .ScrollDidEnd, of: "Scroll")
7
8      // Demonstrate two ways of verifying a clicked link
9      if useMock {
10         let linkHandlerMock = LinkHandlerMock()
11         homeScreen().inject(linkHandlerMock)
12
13         tap("Link")
14         linkHandlerMock.verifyURL("http://www.example.com")
15     } else {
16         perform(tap("Link"),
17                 waitFor: .URLOpened, of: "http://www.example.com")
18     }
19 }
```

Listing 2: UI Test Example: Scrolling & Mocking

First of all, the test case waits for the initial *screen animation* to finish (2).

Afterwards, the screen *title label* is checked, including its availability, visibility and a verification of the displayed text (3).

The next testing step is to *scroll down* to the initially unavailable UI element that is located at the very bottom of the scroll view, while *waiting for the scroll transition to finish* in order to be able to click the link (5–6).

Afterwards, the test splits up in *two branches* which is only done for demonstration purposes, because it does not conform to any best testing practice.

On the one hand, the first branch illustrates an underlying code verification by *injecting a Mock* object[5], while on the other hand the same goal is achieved by making use of the optimized KIF *iOS notification handling*.

Nevertheless, it is usually not very common to test the underlying app implementation within the scope of an UI Test, but it is still the only way to achieve this if no alternative testing process like Unit Testing is involved[6].

As already mentioned, such a basic verification of the underlying functionality can be done by *injecting Mock objects* (10–14). In this specific case, a Mock object is created, which gets injected into the application code afterwards. Therefore, it is possible to *assert against the URL* in order to verify that the correct website would have been opened after clicking the link.

Nevertheless, the default way of achieving such a verification with the optimized KIF UI Testing framework would be to tap the web link, while *waiting for the notification of the URL* afterwards (16–17).

The advantage of this method of verification is that it does only require very *minimal code changes* of the underlying application. This is why this approach is perfectly suitable *for complex iOS apps*, where an according restructuring would not be economically feasible otherwise[7].

Also, the manipulation of the tested iOS application by specifying a special *iOS environment variable* needs to be considered[8]. The reason for this is, that the according iOS app has to be prevented from actually opening any website, because otherwise the UI Test session would get interrupted.

[5]Mocks described in Section 3.5.3
[6]Unit Testing described in Section 2.5.3
[7]Economical aspects described in Section 2.3.2
[8]Environment variables described in Section 5.2.3

Part III.

Outlook & Conclusion

7. Future UI Automation in iOS

This chapter aims at predicting the *short-term future* of the UI Automation of iOS applications. This is why all the described aspects are *only assumptions* and not based on any additional scientific research in the respective field.

Although KIF and Apple's UI Testing are focussing on the same field of application, both are still providing quite *distinctive UI Testing approaches* for iOS apps[1]. Therefore, a potential *coexistence* of these two UI Automation tools is very probable within the next few years.

The main reason for this assumption is that *KIF admits more freedom and flexibility* in comparison to Apple's limited UI Testing interfaces. Moreover, Apple does not seem to be improving the current UI Testing implementation, which is why KIF will potentially *stay competitive* in the near future.

Nonetheless, KIF is split up into *various distinctive implementations* today, closely related to its open-source development nature [34].

As KIF is available for the public use and for further improvement, anybody has the *possibility to contribute*[2]. This is why some of the currently existing separated *optimizations and extensions* will most likely be *integrated back* into the main project, contributing to the open-source community [39].

In fact, even the optimized implementation that has been developed in the scope of this thesis was partly reintegrated into the official KIF project[3].

Nevertheless, *UI Testing* has evolved since the day where it was released in 2015[1]. Furthermore, as it is *deeply integrated* within Xcode and the underlying development kit, it has the possibility to provide a very similar functionality to the optimized KIF version that is described within this thesis[4].

[1] KIF VS UI Testing described in Section 5.1.3
[2] Contributions get verified before being integrated
[3] https://github.com/kif-framework/KIF/pull/869
[4] KIF optimizations described in Section 5.2.3

However, as Xcode has access to all the required iOS functionality, additional application *code adaptions would not be necessary* anymore, which describes a tremendous advantage over all third-party iOS UI Testing tools. Therefore, this official UI Automation solution provided by Apple has theoretically *more potential* than any of its competitors, regardless of their supporting community.

Today, more and more iOS applications are *increasing their minimum target version*, because of the simplified development by omitting the backward compatibility for older iOS devices. This is why the *adoption rate of UI Testing* will increase with a high probability, because plenty of projects can start to make use of it by supporting a minimum target version of *iOS 9 or higher*[5].

In addition to the already mentioned aspects, only Apple's UI Testing automation solution supports the testing of other development *platforms like tvOS or OS X* as well, which indicates yet another very important aspect that has to be considered for its future evolution.

Furthermore, Xcode does only allow to *record UI Test cases* when using its own UI Testing solution[5], which is why solely the testing approach with Apple's default UI Automation tool is able to deliver such a powerful and *time-saving extra functionality*, where not every single UI Test case has to be programmed completely manually.

Finally, due to *KIF's dependency on undocumented Apple APIs*, it cannot be assured that it will always be possible to test with such third-party UI Automation tools in the future. The reason for this is that Apple *could stop to support these APIs*, which would prevent all third-party UI Automation solutions from being able to interact with the UI of an iOS application.

[5] KIF VS UI Testing described in Section 5.1.3

8. Concluding Remarks

This thesis summarized my research in the field of automated UI Testing in iOS. Starting with an introduction to the respective field, followed by Chapter 2 and Chapter 3, covering the topics of software testing in general as well as the subcategory of UI Testing.

After this theoretical part, the technical realization was presented, including an introduction to the specific project in Chapter 4 as well as a summary of the optimized UI Automation process described in Chapter 5. Additionally, more specific implementation details got illustrated in Chapter 6.

Finally, Chapter 7 outlined my prediction of the future of UI Automation in iOS based on an analysis of the currently foreseeable trends.

Appendix

Bibliography

[1] K. Andrews, *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*, Graz University of Technology, Austria, Dec. 2011. [Online]. Available: http://ftp.iicm.edu/pub/keith/thesis/ (cit. on p. xix).

[2] P. C. Jorgensen, *Software Testing: A Craftman's Approach*, 4th ed. Auerbach Publications, 2013, p. 494, ISBN: 9781466560680 (cit. on pp. 7, 9, 10, 13, 27, 30, 33, 36).

[3] D. Gelperin and B. Hetzel, "The Growth of Software Testing," *Communications of the ACM*, vol. 31, no. 6, pp. 687–695, 1988, ISSN: 00010782. DOI: 10.1145/62959.62965 (cit. on pp. 7, 9, 13, 16, 27, 40).

[4] A. Turing, "Checking a Large Routine," in *Report of a Conference on High Speed Automatic Calculating Machines*, AMT, 1949, pp. 67–69 (cit. on p. 7).

[5] A. M. Turing, "Computing Machinery and Intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950 (cit. on p. 7).

[6] K. Beck, *Test-Driven Development: By Example*, 1st ed. Addison-Wesley, 2002, p. 192, ISBN: 9780321146533 (cit. on pp. 7, 13, 31, 85).

[7] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed., December. John Wiley & Sons, 2011, p. 256, ISBN: 9781118031964. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118031962.html (cit. on pp. 8, 13–17, 27, 28, 30–37, 39–41, 67, 80, 86).

[8] G. Gupta and P. Kaur, "Software Testing – Levels, Methods and Types," *International Journal of Advanced and Innovative Research*, pp. 213–216, 2013. [Online]. Available: http://ijair.jctjournals.com/jan2013/t32.pdf (cit. on pp. 8, 33–37).

[9] R. Patton, *Software Testing*. Sams Publishing, 2000, p. 408, ISBN: 9780672319839 (cit. on pp. 8–13, 16–19, 21–26, 28–30, 32–34, 39–41, 79, 81).

[10] H. H. Aiken, "Description of a Relay Calculator (Mark II)," *Annals of the Computation Laboratory of Harvard University, Bd*, vol. 24, (cit. on p. 8).

[11] S. E. T. Committee, "ANSI/IEEE STD 829 – Standard for Software Test Documentation," *Institute of Electrical and Electronic Engineers Computer Society*, 1983 (cit. on pp. 9, 10).

[12] ——, "ANSI/IEEE STD 1008 – Standard for Software Unit Testing," *Institute of Electrical and Electronic Engineers Computer Society*, 1987 (cit. on p. 9).

[13] R. Osherove, *The Art of Unit Testing: With Examples in C#*, 2nd ed., December. Manning Publications, 2013, p. 266, ISBN: 9781617290893 (cit. on pp. 13, 15, 30, 62–64, 73).

[14] R. W. Selby, *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, 1st ed. John Wiley & Sons, 2007, p. 832, ISBN: 9780470148730 (cit. on pp. 19, 22–26, 29).

[15] B. Boehm, "A Spiral Model of Software Development and Enhancement," *SIGSOFT Softw. Eng. Notes*, vol. 11, no. 4, pp. 14–24, Aug. 1986, ISSN: 0163-5948. DOI: 10.1145/12944.12948. [Online]. Available: http://doi.acm.org/10.1145/12944.12948 (cit. on p. 25).

[16] D. Adams, *The Salmon of Doubt: Hitchhiking the Galaxy*, Reprint. 2009, p. 292, ISBN: 9781439568255 (cit. on p. 32).

[17] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.-H. Durand, "Testing Levels for Object-Oriented Software," *22nd International Conference on Software Engineering (ICSE)*, no. 2, pp. 136–145, 2000, ISSN: 0270-5257. DOI: 10.1109/ICSE.2000.870405 (cit. on p. 35).

[18] B. Beizer, *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold Co., 1990, p. 550, ISBN: 9781850328803 (cit. on p. 40).

[19] WG26, *ISO/IEC/IEEE 29119 – Standard for Software Testing*, 2013. [Online]. Available: http://www.softwaretestingstandard.org/ (visited on 08/19/2016) (cit. on p. 41).

[20] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making System User Interactive Tests Repeatable: When and What Should We Control?" *37th International Conference on Software Engineering (ICSE)*, vol. 1, pp. 55–65, 2015, ISSN: 02705257. DOI: 10.1109/ICSE.2015.28 (cit. on pp. 43–45, 47, 49, 51–53, 55, 57, 59).

[21] J. Penn, *Test iOS Apps with UI Automation: Bug Hunting Made Easy*, 1st ed. Pragmatic Bookshelf, 2013, p. 200, ISBN: 9781937785529 (cit. on pp. 43, 49, 53, 57–60, 70, 73, 78, 79, 81, 83).

[22] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, "Mobile Application Testing: A Tutorial," *Computer*, vol. 47, no. February, pp. 46–55, Jan. 2014, ISSN: 00189162. DOI: 10.1109/MC.2013.445. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6693676 (cit. on pp. 43, 49, 51, 57, 59).

[23] S. Arlt, "Program Analysis and Black-Box GUI Testing," no. January, p. 83, 2014. [Online]. Available: https://www.freidok.uni-freiburg.de/fedora/objects/freidok:9425/datastreams/FILE1/content (cit. on pp. 43, 44, 53, 55, 59, 61).

[24] A. M. Memon and M. B. Cohen, "Automated Testing of GUI Applications: Models, Tools, and Controlling Flakiness," *35th International Conference on Software Engineering (ICSE)*, pp. 1479–1480, May 2013, ISSN: 02705257. DOI: 10.1109/ICSE.2013.6606750 (cit. on pp. 43, 45, 46, 51, 53, 57, 59).

[25] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps," *12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 204–217, Jun. 2014. DOI: 10.1145/2594368.2594390 (cit. on pp. 44, 53, 58–61).

[26] M. Özlem and Y. Mete, "Automated Black-Box GUI Testing for Revealing System Bugs in Mobile Applications," *International Journal of Scientific Research in Information Systems and Engineering (IJSRISE)*, vol. 1, no. 2, pp. 65–70, 2015 (cit. on pp. 44–46, 49, 58, 60).

[27] T. Nadu, "Automation Testing Using Coded UI Test," *International Journal of Scientific Engineering and Applied Science (IJSEAS)*, vol. 2, no. 4, pp. 190–194, Apr. 2016, ISSN: 23953470 (cit. on pp. 49, 58).

[28] ABIresearch, *Mobile Application Test Market Boost*, 2012. [Online]. Available: https://www.abiresearch.com/press/200-million-mobile-application-testing-market-boos/ (visited on 08/19/2016) (cit. on p. 57).

[29] M. Kucharski, E. Kolawa, P. Pepek, P. Franczak, J. Labenski, and M. Zielinski, "Dynamically Configurable Test Doubles for Software Testing and Validation," 20 150 378 880, Dec. 2015. [Online]. Available: http://www.freepatentsonline.com/y2015/0378880.html (cit. on pp. 63, 64).

[30] Apple Inc., *Apple iOS 7 Release Notes*, 2013. [Online]. Available: https://support.apple.com/kb/dl1682 (visited on 08/19/2016) (cit. on p. 67).

[31] G. Lodi, *Xcode 7 UI Testing, A First Look*, 2015. [Online]. Available: http://www.mokacoding.com/blog/xcode-7-ui-testing (visited on 08/19/2016) (cit. on pp. 73, 75).

[32] J. Sherman, *UI Testing in XCode 7, Part 1: UI Testing Gotchas*, 2015. [Online]. Available: https://www.bignerdranch.com/blog/ui-testing-in-xcode-7-part-1-ui-testing-gotchas (visited on 08/19/2016) (cit. on pp. 73, 75).

[33] E. Firestone, *iOS Integration Testing*, 2011. [Online]. Available: https://corner.squareup.com/2011/07/ios-integration-testing.html (visited on 08/19/2016) (cit. on p. 74).

[34] *Keep It Functional – An iOS Functional Testing Framework*. [Online]. Available: https://github.com/kif-framework/KIF (visited on 08/19/2016) (cit. on pp. 74, 78, 80, 81, 87, 97).

[35] *Xcode 7.0 Release Notes*, 2015. [Online]. Available: https://developer.apple.com/library/ios/releasenotes/DeveloperTools/RN-Xcode/Chapters/xc7_release_notes.html#//apple_ref/doc/uid/TP40001051-CH5-SW29 (visited on 08/19/2016) (cit. on p. 75).

[36] *Apple User Interface Testing*. [Online]. Available: https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html (visited on 08/19/2016) (cit. on p. 75).

[37] D. Knott, *iPhone Test Automation Using KIF*, 2011. [Online]. Available: http://adventuresinqa.com/2011/12/02/iphone-test-automation-using-kif-keep-it-functional (visited on 08/19/2016) (cit. on p. 78).

[38] *KIF Accessibility Handling: Identifier VS Label*. [Online]. Available: https://github.com/kif-framework/KIF/issues/243 (visited on 08/19/2016) (cit. on p. 78).

[39] J. Suliga, *UI Automation: Keep It Functional and Stable*, 2016. [Online]. Available: https://engineering.linkedin.com/blog/2016/01/ui-automation--keep-it-functional--and-stable- (visited on 08/19/2016) (cit. on pp. 80, 82, 97).

[40] P. Steinberger, *Running UI Tests on iOS with Ludicrous Speed*, 2016. [Online]. Available: https://pspdfkit.com/blog/2016/running-ui-tests-with-ludicrous-speed (visited on 08/19/2016) (cit. on p. 83).

[41] Realm Inc., *SwiftLint: Static Code Style Analysis*. [Online]. Available: https://github.com/realm/SwiftLint (visited on 08/19/2016) (cit. on pp. 86, 87).

Acronyms

The following Table 8 gives an overview of all the acronyms used throughout this Master's Thesis. However, all indicated acronyms are also defined and described at the place of their first occurrence within the Thesis.

Acronym	Meaning
TDD	Test-Driven Development
UI	User Interface
GUI	Graphical User Interface
IEEE	Institute of Electronics and Electrical Engineers
ISTQB	International Software Testing Qualification Board
API	Application Programming Interface
KIF	Keep It Functional
CI	Continuous Integration

Table .1.: Used Acronyms Overview