University of Bremen
Faculty 3 – Mathematics and Computer Science
Master Program of Digital Media

Volkswagen Group AppFactory

Master Thesis

Approach for the automated testing of design requirements on different versions of mobile devices

Name: Nadezda Bogdanowa
Matriculation number: 2302623

1st Supervisor: Prof. Dr. Gabriel Zachmann
2nd Supervisor: Prof. Peter von Maydell
Mentors: Rainer Riekert
Ingo Wolterstorff

Bremen, May 2014

Statutory Declaration

I declare that I have developed and written the enclosed Master Thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Master Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

……………………………… ……　　　　　　　　　……………………………………………
date　　　　　　　　　　　　　　　　　　　(signature)

Abbreviations and Acronyms

UI	User Interface
GUI	Graphical User Interface
CI	Corporate Identity
CD	Corporate Design
OS	Operating System
API	Application Programming Interface
IDE	Integrated Development Environment
SDK	Software Development Kit
XML	Extensible Markup Language
XAML	Extensible Application Markup Language
QML	Qt Meta Language or Qt Modeling Language
XIB	XML Interface Builder
MVC	Model-View-Controller
ALM	Application Lifecycle Management

Table of Contents

1. Motivation 8
2. Related work 10
   2.1. Mobile operating systems 10
   2.2. Testing classification 12
   2.3. State of the art 16
3. Background 23
   3.1. Internal structure of Volkswagen Group AppFactory 23
   3.2. Corporate Identity and Corporate Design 24
4. Defining the design guidelines for mobile devices 26
5. DesignTesting Framework: Approach for the automated design testing tool 31
   5.1. Scientific question 31
   5.2. Different ideas 31
      5.2.1. Source code analysis of layout files 31
      5.2.2. Source code analysis of the application code 33
      5.2.3. Screenshot analysis through image recognition tools 33
      5.2.4. Screenshot analysis through image comparison 34
      5.2.5. Combination of various methods 34
   5.3. Approach for automated design testing tool 35
      5.3.1. General idea 35
      5.3.2. Physical constraints 35
      5.3.3. Functional overview 36
      5.3.4. Architecture 40
   5.4. Examples 50
      5.4.1. iAgree 50
      5.4.2. Konzernkalender 53
   5.5. Limitations 56
6. User study of DesignTesting Framework 57
   6.1. Analysis of target group 57
   6.2. Procedural method 57
   6.3. Experiment design 58
   6.4. Data representation 61
   6.5. Data analysis 65
   6.6. Summary of the results 68
7. Conclusion 70
8. Future work 72
9. Acknowledgements 75
10. References 76

List of Figures

Figure 1 Model-View-Controller …………………………………………………………….... 11
Figure 2 Comparison of architecture of Android and iOS operating systems …………..... 12
Figure 3 The flow of the testing process ……………………………………………………. 17
Figure 4 Volkswagen company and product logotypes ……………………………………. 25
Figure 5 Java implementation of the layout files source code analysis approach ……..… 32
Figure 6 Structure of the automated design testing tool …………………………………... 36
Figure 7 Data flow of DesignTesting Framework …………………………………………... 37
Figure 8 Screenshot of user preferences settings ………………………………………..… 38
Figure 9 Screenshot of storing data with iTunes function …………………………………. 38
Figure 10 Implementation of dumpSubviews function ……………………………………… 42
Figure 11 Implementation of check consistency function …………………………………… 43
Figure 12 Implementation of image comparison function ………………………………….. 46
Figure 13 Implementation of pixel by pixel comparison method …………………………… 46
Figure 14 Implementation of reading the design requirements function ………………….... 48
Figure 15 System method architecture ……………………………………………………….. 49
Figure 16 Screenshot of iAgree main view with accept and decline options …………….... 51
Figure 17 Screenshot of iAgree view with opened vertical slider ………………………….. 51
Figure 18 Design requirements for iAgree presented in the Excel table ………………….. 52
Figure 19 Results of the design testing of iAgree …………………………………………… 52
Figure 20 Screenshot of Konzernkalender year view ……………………………………….. 54
Figure 21 Screenshot of Konzernkalender month view …………………………………….. 54
Figure 22 Design requirements for iAgree represented in the Excel table ……………… 55
Figure 23 Results of the design testing of Konzernkalender ………………………………. 55
Figure 24 Results of the usability questionnaire …………………………………………….. 63

List of Tables

Table 1 General design requirements 27
Table 2 Volkswagen corporate design requirements 30
Table 3 Test cases for the research experiment 61
Table 4 Duration of every test case for both manual and automated design testing 62
Table 5 Number of errors for every test case for both manual and automated design testing 62
Table 6 Usability questionnaire with average results 64
Table 7 Experiment results of test person #1 66
Table 8 Experiment results of test person #2 67
Table 9 Experiment results of test person #3 68

Summary

With the extensively increasing number of mobile applications on the market, an automating and effective testing of applications has become a relevant research challenge. Among different types of testing, the evaluation of the graphical user interface (GUI) is one of the core issues. The testing of the GUI shows whether the UI elements are correctly displayed on the screen and how well the user can interact with them. For the development of applications for companies, not only the functionality and the response of the application to the user's actions is important, but also the compliance of corporate identity, general design guidelines and customer needs concerning the visual appearance. Accordingly, the verification of the visual representation of the GUI is an important part of the GUI testing. However, it is difficult to identify the design requirements with a human eye as precisely as the computer would do this. So in Volkswagen Group AppFactory the design testing is either conducted by human testers, through manual verification of the required components, or not conducted at all because of the lack of required technology. An automation of the design testing process can improve the results and reduce effort, time and the cost of the testing process. In recent years a number of diverse automated tools for the GUI testing were presented. Nevertheless, the most currently existing techniques were developed only for the testing of functionality, safety or usability of the application, not taking into account the layout design. No automated tool for the verification of the design requirements and corporate design of mobile applicationsis currently known.

This master thesis presents a prototype of the automated testing tool, called DesignTesting Framework, for the evaluation of design requirements of iOS applications. My framework, developed in Objective-C, can be linked within XCode project to every iOS application with the available source code. While running the application on the mobile device, the DesignTesting Framework can be activated through the shaking motion of the device. My automated design testing tool goes recursively through the source code of the application, finds all UI elements presented on the screen, and defines their attributes. After that, the tool reads the customer and corporate design documents and compares the design requirements with the values in the application. Finally, the tool represents the test results in the structured PDF document. The DesignTesting Framework is able to verify the general design guidelines, the corporate design, and the requirements of the customers concerning the visual appearance of the application. I have created the list of design requirements appropriate for the mobile applications, since most existing guidelines for the development of the GUI cannot be applicable for handheld devices. These guidelines are also presented in my master thesis.

The evaluation of my DesignTesting Framework in a user study shows that it can reduce the time of the design testing process, especially using the tool for the large tests. The most significant outcome of the evaluation demonstrates that the use of my automated tool leads to more accurate and precise testing results, almost without errors, while the manual design testing causes a large number of errors. However, some improvements can be done in future work to raise the effectiveness of the DesignTesting Framework. The users agree that the invention of my automated tool in the design testing process can reduce production costs, increase productivity and will lead to the development of more qualitative and visually appealing applications.

Approach for the automated testing of design requirements on different versions of mobile devices

1. Motivation

Mobile applications have become very popular in the last few years and are in fast development [37]. A mobile application is defined as a type of software application considered to run on hand-held devices, such as smartphones or tablet computers. They can be designed for the same tasks as those running on desktop computers, but due to their small size and limited resources the mobile applications often focus on certain isolated functions.

The growing number of mobile applications on the market has attracted an increase in research interests in this field. According to the Portio Research, 1.2 billion people were using mobile applications at the end of 2012 [25]. In 2013 almost 2 million mobile applications were available for download at four leading app stores – Apple (900 000 apps), Android (800 000 apps), Blackberry (120 000 apps) and Windows (100 000 apps).More than 100 billion applications were downloaded by the end of 2013 - 48 billion Apple applications, 50 billion Android applications and 3 billion Blackberry applications [24].

With the exponential increase in mobile application development, an effective testing of them has become a relevant research challenge. However, the methods and guidelines traditionally used in software testing may not be applicable for mobile applications because of the differences and limitations of hand-held devices. The most significant differences of mobile devices include mobile context, connectivity, small screen sizes, different display resolutions, limited processing capability, and power and data entry methods [49]. While iPhone and iPad have a small number of screen types, Android is running on a variety of devices with different resolutions. Apart from this, various operating systems have differences not only in the code architecture, but also in the visual appearance of native UI elements. So every mobile device can react differently to the same code. All these differences should be taken into account by designing the mobile application as well as by testing it.

The effective testing of mobile applications is an "emerging research area that faces a variety of challenges due to unique features of mobile devices" [49]. Currently there are several studies and approaches regarding the testing of the functionality, security and usability of mobile applications, but the field of GUI design testing has still not been widely explored. This thesis concentrates on the design testing, which is the part of usability, though it focuses not on how well the users can interact with the application, but rather on the visual appearance of the GUI. The challenge for GUI testing is to verify whether native applications are correctly displayed on different devices.

An automation of this testing can reduce time, effort, errors and the cost of the testing process, and increase productivity. The automation of the software testing process has numerous benefits, which are described by Melody Y. Ivory in "The State of the Art in Automating Usability Evaluation of User Interfaces" [19]. The most important advantage is that of cost-saving, due to the reduction in the testing time. Another positive point is the prediction of the time and error expenses through the whole application. An additional

benefit is the expansion of the tested features. The use of automated tools makes it possible to cover all possible test cases and user interactions, something that is not always achievable with non-automatic testing. Apart from this, the special tools not only perform automatically the test cases and simulate the user interaction with the system, but are also able to expertly analyze the obtained results. Not all testers have enough competence in all aspects of software evaluation. One more advantage is the possibility of the comparison between optional designs. During the manual testing only one designed UI is evaluated. Some automated tools make it possible to predict and simulate the alternative and improved designs and to test them. Finally, the automated tests can also be performed during the development phase, with the UI schemes, prototypes and guidelines predicting the bugs before implementing them. The human testers as a rule test only the implemented version of the UI.

Initially, most mobile applications were developed for entertainment purpose, but now many industries have arranged application development for competitive benefit. In order to support the image of the companies their applications should meet certain design requirements and follow the corporate identity. The compliance of the corporate design is very important for company identification, maintaining its image in the media and for the consistent appearance of the application.

The current situation shows that there is a need of an automated testing tool that can evaluate the visual design of the mobile application, according to the general design guidelines and the requirements of customers and the company. Such an automated design testing tool can improve and simplify the process of testing the user and corporate design requirements.

The goal of this master thesis is to present an approach for the automated GUI testing of the user and corporate design requirements. The result of this work is a prototype of the automated design testing tool for iOS applications, developed for Volkswagen Group AppFactory. The thesis includes an overview on existing systems and different methods of mobile testing, a general idea of a unified design testing tool for different devices, a presentation of the design testing approach on the example of iOS application, testing and evaluation of this approach, as well as a discussion of the challenges and future research questions in the field of automated design testing of mobile applications.

2. Related Work

2.1. Mobile operating systems

In order to test the application design on different devices, it is necessary to be aware of the structure, source code and UI representation of different platforms. A paper, "Cross-Platform Mobile Application Development: A Pattern-Based Approach" [1], describes how most applications are build. Most of the mobile platforms use the Model-View-Controller principle, which is demonstrated in figure 1. It consists of three parts, which are responsible for the core logic (Model), visual representation (View) and the interaction with the user (Controller). This principle allows one to separate functionality and layout design of the application. So while testing the layout, it is possible to consider only the design of the application, independent of the functionality behind it. However, sometimes the design is described in the source code. In particular, the correlations between views and elements cannot always be recognized in the layout files. The authors describe various patterns for the individual screen representation and give a detailed overview on UI elements of different platforms. Most core UI elements of various platforms are similar. By understanding the common components and their differences, it is possible to develop an approach for the design testing of applications.

The layout description results in the graphical GUI, which is the main object of this thesis. A description of the GUI is given in [3]: "A GUI provides a hierarchical, graphical front-end to the application. It is usually implemented as an event-based system that accepts as input user-generated events and produces graphical output. Each GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI."

With the large number of different mobile devices and operating systems, it is important to consider the differences in their GUI elements and resolutions. The paper, "Orientation Awareness in Declarative User Interface Languages for Mobile Devices: A Case Study and Evaluation" [22], describes the differences in various screen resolutions, how the layout is fitted onto them and how the UI elements are resized and redesigned according to the mobile platform. Furthermore, the authors describe the declarative languages used in different platforms for describing the layout information. Many of the platforms use XML based languages (except for Qt Modeling Language QML), which define a hierarchical structure of GUI elements and their attributes. Figure 2 shows the tree structure of the core UI elements of Android and iOS applications.

A paper, "Evaluation of Descriptive User Interface Methodologies for Mobile Devices" [36], also explores the most popular operating systems and their UI descriptions. The Android operating system developed by Android Inc. and acquired by Google was released as a smartphone platform in 2008. It is Linux-based and uses Java language for the development. The UI description is defined using XML-based declarative language and has a hierarchical structure. It consists of multiple views and elements, having the correlating layout type. UI design can also be described programmatically using Java API. The connection between different screens is defined in the source code and cannot be read from the XML-file.

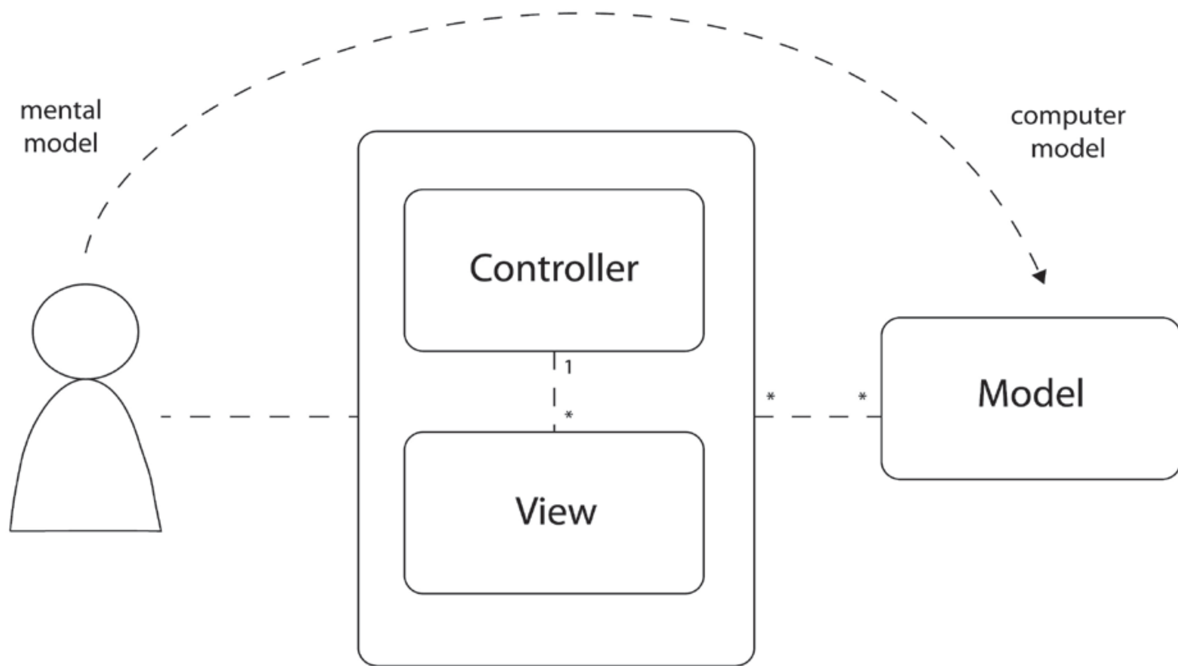

Figure 1: Model-View-Controller

iOS is the iPhone platform presented to the market in 2007. The native language for iOS development is Objective-C. For the UI description, iPhone uses the Interface Builder based on the WYSIWYG principle, but the output is stored in the XIB file, which is XML-based and hierarchically organized. During building, the XIB files are compiled into NIB files that cannot be edited by a hand. The connection between screens is similar to Android architecture and is defined programmatically in the source code.

Windows Phone 7 OS is built on CE kernel and uses .NET languages (C# and VB .NET) for the application development. It uses XML-based declarative language XAML for the UI definition. Each interface element described by XAML is a .NET object. The relations between screens are made in the source code with a target on the correlating XAML file. This way the Windows UI creation is very similar to Android. Nevertheless, there is an opportunity to create these connections inside the XAML file itself.

The BlackBerry OS provides two possibilities for the application development – with BlackBerry web development using widgets, and with Java application development using MIDP 2.0. For the creation of this, the UI BlackBerry offers a framework, Cascades, based on Qt. It uses a specialized markup language QML, which is not XML-based. QML uses JavaScript for creating the UI elements and connections between them, so it is possible to develop the complete application using QML, though it is initially thought for the UI definition.

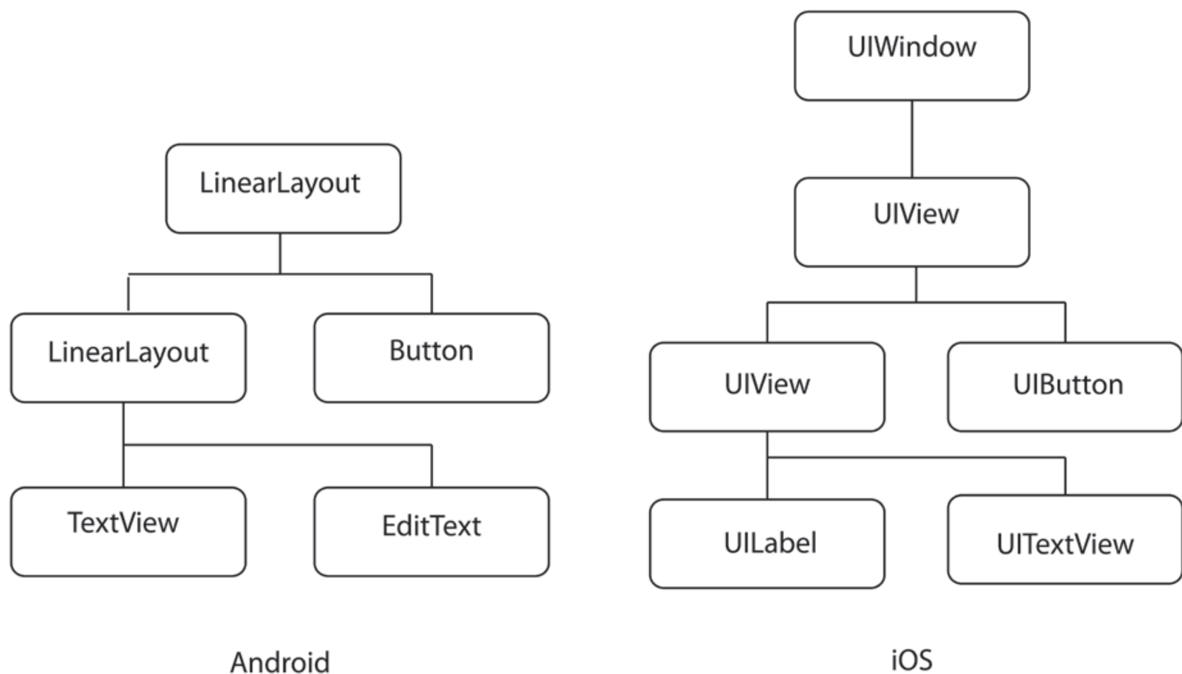

Figure 2: Comparison of architecture of Android and iOS operating systems

2.2. Testing classification

The fast growth of the mobile application market requires specialized testing techniques, because of the high defect density and the new kinds of bugs caused by the physical limitations of mobile devices. There are many different types of automation tools, based on the testing purpose, automation type, technique and the availability or unavailability of the code.

Concerning the mobile application components that differ from the desktop software, and the need to be tested, Henry Muccini et al. categorize in “Software Testing of Mobile Applications: Challenges and Future Research Directions” [27] different types of automated tests as following:

- Mobile connectivity testing

The network connection of mobile devices is one of the most critical and unstable characteristics that can vary in different situations, and as a result affect the functionality of the application. So the mobile application should be tested for reliability, performance and functionality through different networks and in different connectivity conditions.

- Testing of limited resources

The distinctive characteristic of mobile devices is the limitation in their resources. Even the most powerful devices cannot be compared with laptop and desktop computers. Therefore, the resource usage of the application has to be tracked to find out whether it limits the performance of the application and needs to be shortened.

- Autonomy testing

Another constraint of mobile devices is the high energy consumption, which is also different for every device and in different usage scenarios. So the energy consumption of the application can be tested through monitoring it while running the application on different devices and in different situations.

- User interface testing

The screen of mobile devices has a smaller size and resolution in comparison to laptop and desktop displays, so the design of the GUI needs to follow special guidelines. Dependent on the various mobile devices, the GUI can react differently to the source code. It is necessary to test how the UI components are displayed and how well the user can interact with them.

- Context awareness testing

Many mobile applications use sensors or connectivity options for their functionality, such as noise, light, motion and image sensors, as well as Bluetooth, GPS or Wi-Fi. This causes the large flow of data that can vary in different environmental contexts. The testing of the functionality in various environment situations, and in combination with different contextual inputs, is the important aspect of the evaluation of the context-dependent applications.

- Adaptation testing

The mobile applications may sometimes need a runtime adaptation to some contextual information. So testing the application on the adaptation correctness is an important challenge.

- Testing of new programming languages

For implementing and designing applications, new programming languages and new libraries optimized for mobile devices were invented. The usual byte code and structure analysis techniques cannot always be applicable for new mobile languages and need to be evaluated.

- Testing of new operation systems and diversity of devices

There are different operating systems available for mobile devices, and also new versions of them regularly appear. Some functional bugs may be caused by OS problems. That's why mobile applications have to be tested on OS compatibility and diversity coverage.

- Input possibilities and touch screen testing

Unlike desktop computers with the mouse and keyboard as input modalities, most mobile devices use touch screens as the main input possibility. The response time of the touch screen depends on the device and on the current state. The reaction of the system to the touch input needs to be tested on different devices and under different conditions, such as resource usage, processor load and memory load.

The objective of this thesis is to develop an approach for the design evaluation, which is a part of the GUI testing. It is one of the most important parts of the application's evaluation, since GUI is a core environment, which is used to the system's interaction with the user. GUI testing is a process to evaluate how the UI components are rendered to the screen and how they react to the user's input [5] [38]. GUI testing includes a wide range of tasks, for example evaluating how the UI handles the touch events and input data, how it displays different UI components, such as buttons, menu items, toolbars, text boxes, list controls, images and others, and how it responses to the interaction with these components. Commonly, users perform manually different actions with the application several times with different data input and compare the result with the expected behavior. GUI testing can prevent a large number of problems from small bugs to the total crash of the system. GUI testing includes testing of functionality, that checks how the system responds to the user actions, the usability, that checks not only if the application is functionally correct, but also if it is easy to use, and the design testing, that checks the correct and attractive visual representation of the UI components. Design testing includes the verification of whether the GUI components were correctly rendered to the screen with the expected color, style, size and position attributes.

In general, according to Balbo [6] [19], all evaluation methods of the listed components can be divided into four groups concerning the automation type:

- Non-automatic or Manual

This method involves no level of automation and is entirely performed by the human testers.

- Automatic Capture

The capture method collects and records various software data, such as keyboard and mouse input, visual output, task completion time, errors, guidelines non-compliance and other things for the future test regeneration or analysis.

- Automatic Analysis

The analysis method automatically interprets the usability output and determines the bugs and potential problems, depending on the test results.

- Automatic Critic

The critic method not only finds the problems of the system, but also offers their possible solutions in order to improve the application performance.

This thesis focuses on the automatic testing of the applications, especially on the automatic analysis and the critic one. Automatic analysis of the GUI is the use of the special tool that can automatically simulate the user interaction with different GUI components of the application and analyze if the application responds in the way it is supposed to. Automated GUI testing compares whether the outcome of the test corresponds to the expected results, and finds out the problems in the GUI. It can replace the manual input, reduce the testing

time and input errors, and find the problems that sometimes cannot be identified manually. The automated tools sometimes also display the expected results, showing what components of the application must be changed.

In recent years a number of different automated tools for the evaluation of applications has appeared. They can all be classified in some categories. According to the tester's knowledge of the inner workings of the system under test, there is a separation between white-box and black-box testing [14] [20] [40]. White-box testing is a technique of the structural and logical analysis of the application source code, control flow and data flow. It requires access to the application project or to the compiled code. Black-box testing does not need to have access the source code. It analyses the functionality of the application through interacting with it. The white-box technique is often referred to as structural testing and black-box as functional testing. The combination of both techniques with the limited knowledge of the internal workings of the system under test is defined as grey-box testing. This thesis considers both the white-box and black-box testing possibilities, but the final approach requires access to the application's project and is based on the analysis of the source code during its compilation. Therefore my approach uses the white-box technique.

Another possibility to classify the testing techniques, with reference to the state of the system under test, is a separation between static and dynamic analyses [42]. Static analysis tests the application in a non-runtime environment. Commonly, it examines the source code of the system, but it can also be applicable for the black-box testing, e.g. reviewing the visual representation of GUI without interacting with it. Dynamic analysis examines the performance of the application while running. Typically, dynamic analysis is used in black-box testing, but it can also be used while executing the source code. My prototype uses the dynamic analysis, since it evaluates the GUI of the current application view, so that every possible state of the application can be tested. It analyses the source code at the runtime of the application in order to find the UI components that are currently present on the screen.

In addition, Hughes Systique Corporation suggests in the paper "Test Automation Tools for Mobile Applications: A brief survey" [14] another classification of the currently available automation tools:

- Platform specific

There are tools provided by the operation systems and integrated in the SDK development environment, for example Instrumentation and MonkeyRunner for Android or Instruments for iOS. They can test the application during the development process within the IDE.

- Generic script based

Some tools such as Sikuli or Robotium can control the performance of applications using scripts. The user needs to write the test description with the required script language and to perform the test through executing this script.

- Random event generator

There are also tools that are able to send random events to the application to simulate the user interaction and to repeat the same user actions multiple times. The examples of random event generator tools are Monkey and iOS Automation.

These techniques show that the testing of mobile applications is a broad field covering a diversity of methods. In my thesis I focus on the automatic analysis and critic of the GUI, particularly on design testing. For this purpose, I will consider both white-box and black-box possibilities, based on the code and layout analyses, and will expand my approach to white-box dynamic testing.

2.3. State of the art

Currently, the usual process of application testing is based on the creation of scenarios and test cases, and going through them manually [16]. Therefore, they are based on the manual interaction between the tester and the system and on the visual analysis of the obtained results. The tester records his observation about the behavior of the system under test and marks the test as passed or failed. This process is shown in figure 3. Therefore the main goal of test automation is to reduce the person's interaction with the system to save time in human resources.

An automation of testing can reduce effort, time and the cost of the testing process, because more tests can be run in less time and human resources are free to perform other manual tasks. In addition, it can improve testing efficiency because the human may not always find all defects. The testing in same conditions is also important for the consistent and repeatable testing process and clear and comparable test results [14].

Despite the novelty in the field of mobile development, the need for automation is evident. Therefore many solutions in the UI analysis have been already presented in recent years. This chapter presents several existing tools and approaches for the automated testing of mobile applications. A number of different currently available automated tools is described in “Test Automation Tools for Mobile Applications: A brief survey” [14]. Some of them belong to the standard tools that are used as a basis for more complicated approaches.

For example, the Android Instrumentation is a framework provided by operating system and integrated in the Android SDK. It is based on the JUnit framework and requires access to the source code of the application. Instrumentation examines the GUI behavior produced by user interactions and fired events.

Instrumentation is abstracted in Robotium, which enables the preparation of grey-box automated test cases for applications. Robotium can be used both for applications with the source code available and without code information. With a help of Robotium, it is possible to write function, system and acceptance test cases, to find current activities and views and to make decisions automatically.

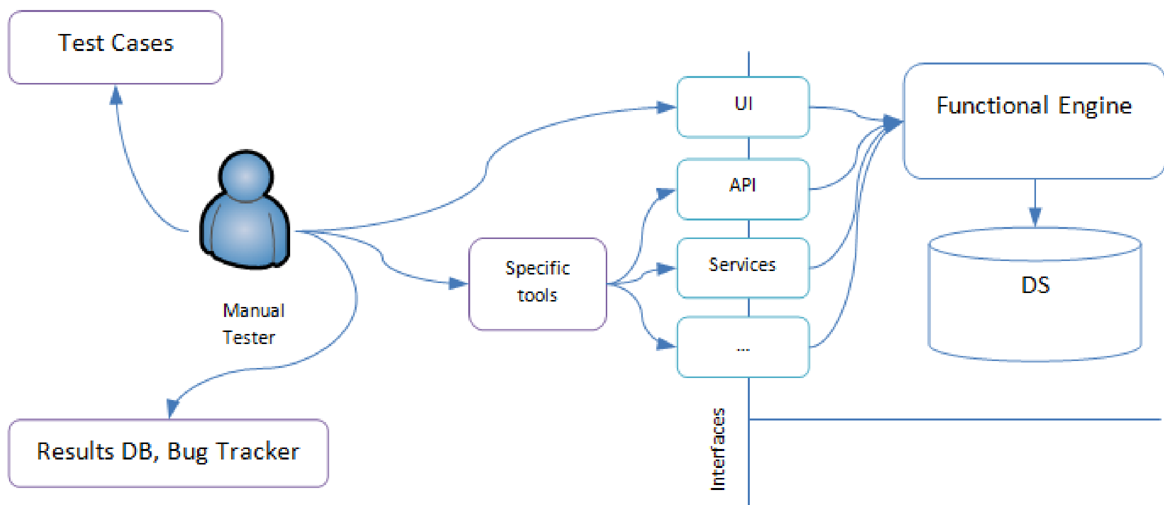

Figure 3: The flow of the testing process

Another platform-specific system included in Android SDK is Monkey [7], including Monkey tool and MonkeyRunner. Monkey tool is running directly on the mobile device and allows the generation of random events, such as key presses or screen touches, in order to discover the potential bugs by searching the known error patterns. MonkeyRunner is an API build on Monkey tool that enables functional testing and requires writing Python scripts to manage the testing process. It allows the sending of key events, taking screenshots of GUI and programmatically controlling the testing process on multiple Android devices at the same time. MonkeyRunner can compare screenshots with reference images to validate the visual correctness of the GUI.

There is also a platform-specific automated analyzing tool for iOS applications called Instruments. It includes an UI Automation template and UI Automation API for performing different test cases on the device. Users can write custom Java scripts to simulate the user interaction for a certain time period and record events, gestures, and data information. In addition, UI Automation allows testers to investigate whether the mobile application is performing according to the user's expectations. The tool represents the hierarchy of UI elements and their values, and analyzes the returned information. UI Automation can be easily integrated with other templates of Instruments to test different aspects of the application functionality.

Android apktool is a reverse engineering tool for Android applications that can reconstruct the binary code and resources to nearly original form. Hierarchy Viewer for Android applications can also deconstruct the GUI and represent its hierarchical layout structure in a visual form. iOS Hierarchy Viewer provides the similar functionality for iOS applications. It debugs the UI and displays the UIViews hierarchy and the property values.

Introspect is a library for iOS, supporting the debugging of the application's layout [35]. It is particularly beneficial for the dynamically created and changing UIs. It displays the

information about GUI elements, such as views location, size, actions and targets, and enables the layout change during runtime. In addition, Introspect creates and outputs the view hierarchy model.

GUITAR is a framework for Android, which enables the automated testing of applications GUI. It includes platform-specific GUITAR Ripper and Replayer, and is based on reverse engineering and model-based techniques. The testing process with GUITAR involves four phases: ripping, which interacts with UI and represents the structure and relationships between GUI components, model construction, which extracts the GUI model and generates an event flow graph, test case generation for the automated testing process and replaying, which analyses the test cases' results.

The study of Robin Goldberg et al., presented in „Automatisierte, quantitative Analyse von Android-Applikation-GUIs" [11], evaluates the reverse engineering technique in the testing of mobile applications. For this purpose, the authors implemented a testing tool APKAnalyzer, which can read the resource folder of the application. This folder contains the resource files required for the GUI, including images, animations, XML-files and manifest file. Extracting these files, the tool then analyses the following information: activities, uses-libraries, uses-permissions, uses-features, uses-sdk, hardware acceleration, themes and UI options. For every found reference and activity, APKAnalyzer searches for the correlating layout files through analyzing the manifest and SMALI files. The layout files contain the information about GUI elements, though sometimes the layout information can be described in the source code.

One of the possible solutions for automated testing is the reverse engineering technique. "Reverse engineering is the process of analysing a subject system to create representations of the system at a higher level of abstraction" [26]. This method, used for desktop applications, is described by Ines Coimbra Morgado et al. in "Reverse Engineering of Graphical User Interfaces" [26]. The authors implemented an automated model-based testing tool called ReGUI. It is developed using the UI Automation framework, which represents all opened applications in a tree structure. The aim of ReGUI is to reconstruct the application structure while interacting with it, so it is based on the dynamic testing approach. The tool goes through all menu options of the application under test, determines which GUI elements are enabled and which are disabled, and interacts with all enabled elements. After that, ReGUI closes all opened windows and repeats the process of navigating through the menu again. During this procedure it determines whether any element state has changed. As a final point, it creates an output representing the structure and behavior of the GUI. It includes six files: ReGUI Tree, Window Graph, Navigation Graph, Disabled Graph, Dependency Graph and Spec# file.

This technique can also be applied for mobile application testing. In the papers: "Using GUI Ripping for Automated Testing of Android Applications" [2] and "A Toolset for GUI Testing of Android Applications" [3], Domenico Amalfitano et al. presented a similar analyzing approach but for the mobile platform Android. AndroidRipper is a dynamic model-based automated tool for analyzing the GUI of the Android application, using the reverse engineering technique. The purpose of the tool is ripping and reconstructing the GUI structure and searching for failures through interacting with the application under test. This ripping method is developed using Robotium Framework and Android Instrumentation class. The tool includes nine modules - Scheduler, Robot, Abstractor, Extractor, Engine,

Strategy, Planner, Comparator and Persistence Manager. The whole process consists of three steps: deploying, ripping and creating an analyzing report. AndroidRipper defines an initial state, generates new events and iteratively executes the GUI structure. During the ripping process, the tool finds the application's initial state, the fired and fireable events and state transitions, and creates a hierarchical state machine model of the GUI. Finally, it analyses the model, observing the resulting GUI state changes, and determines the failure of the application under test. The output of the system is a XML file with the GUI Tree and a crash report file.

Another opportunity is presented in the paper: "Insights into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps – Source code analysis" [33]. The authors describe an automated analyzing approach for Android applications through decoding and reconstructing the source code and resource folder. For this purpose, the authors use apktool to find the layout files, images, string files and other resources. They analyze the UI, considering the graphical objects, UI layout files (XMLs) and different screen resolutions. In addition, it is possible to read the manifest file and to determine the number of activities and XML files corresponding to each activity through finding the necessary functions and a layout file ID. With the help of this method, the authors of the paper have analyzed the user interface elements and design patterns of various applications.

One more technique for automating testing of GUI of Android applications is described by Cuixiong Hu et al., in "Automating GUI Testing for Android Applications" [13]. This white-box approach detects the GUI bugs using random test case and event generation. It determines all activities in the application, runs created test cases through the Dalvik Virtual Machine to simulate the user interaction, feeds events to the application, finds out the required information (GUI events, method calls, and exceptions), writes it in the system log file and evaluates this file searching for the GUI failures. The tool identifies whether the activity was created correctly, whether it executes according to the GUI specification, and whether the application can properly enter and exit the state. For the test case generation, the authors use the Activity Testing class in JUnit, and for the event generation they use Monkey tool.

In contradiction to dynamic analyzing technique, static analysis examines only one state of the application not interacting with it. The example of this method is described by Atanas Rountev et al. in "Static Reference Analysis for GUI Objects in Android Software" [30].This approach is based on defining the flow of node objects and their relation to activities. In order to imitate the run-time performance of the application, the authors define the hierarchical model of GUI objects and objects interactions with activities, listeners and other views through the static analysis of object references. These interactions identify the potential GUI events at each state of the application. In object-oriented Android applications each event handler is related to a certain GUI object, provided with a unique object id. Defining these connections, it is possible to recognize the flow of references, controls and data. The procedure consists of three phases: creating the constraint graph of GUI objects, analyzing all executable methods in the application, computing the object relationships, and propagating views through the constraint graph, based on the recognized relationships. The reference analysis is possible for object-oriented languages, so it can be applicable not only for Android but also for Objective-C-based iOS applications.

An approach for dynamic interaction with a GUI of iOS applications is presented in “Challenges for Dynamic Analysis of iOS Application" [35]. Martin Szydlowski et al. developed a prototype that interacts with the GUI and analyses its components. To execute the system on the application they use a VNC server and a modified VNC client python-vnc-viewwer4. The tool calls Objective-C methods at a runtime, defines their arguments and creates the list with the information about detected methods. Finally, the tool sends the received information in the form of the script file to the analysis report and examines the methods functionality. Furthermore, in order to identify interactive UI components, the tool uses image comparison between the application’s current state and the previous one. The change in large numbers of pixels between two screens means that the interaction was performed. To find the element being fired, the tool sends tap events to every location of the previous screen and observes if it changes to the reference screen.

It is also possible to use the combination of static and dynamic analysis, which is described by Cong Z et al. in the paper “SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Application” [50]. This method is used in a testing tool, SmartDroid, and includes two phases. The first one is a static analysis, which finds the potential activity switch paths from the Activity and Function Call Graphs, using a depth-first searching algorithm. The second phase is the dynamic analysis, which is applied for each found activity and navigates through all GUI elements and examines their interactions. The dynamic analysis stage includes three components: UI Interaction Simulator, Activity Restrictor and Runtime Execution Environment. SmartDroid is based on apktool and uses the shell and Python scripts.

Another method for automated design verification of the system was invented by David Todd Massey et al. [23]. Like the previous techniques, it also tests the design structure of the system and the references between objects and not the visual representation. The specialty of this tool is that it verifies the schematic design description created by the design simulating programs and not the final GUI. It interprets the design description as a state diagram and creates the objects and the relations between them. In this way, it simulates the design of the system’s interface from it schematic representation. After that, the tool displays the output data from the simulated structure and decides which elements of it should be tested. A test generator sends events to the simulated design in order to execute these elements and their connections. Then the tool reports the elements that have not been tested yet and generates new events to exercise them. The output data of the simulated design from the first execution is compared with the output data of the references in the second execution. The tool associates the received results with the initial objects to define if any of them were performing incorrectly. So the bugs in the design of the system can be found already from its description before it will be implemented.

Most of presented methods use the similar technique of testing mobile applications through creating the model with hierarchical structure and analyzing the flow of the potential GUI elements, their references, and method calls. There is also a different method based on the functional testing of GUI using screen capture and visual recognition in order to find a certain pattern on the screen and interact with it.

An automated software testing system invented by John A. Gregory et al. [28] can be used to automate the testing and to compare the design of different versions of the system. During the execution of the system under test, the tool records all inputs, such as keystrokes and mouse events, and saves them in the script. Furthermore, it captures the

screen images of the system. When the next version of the system under test is executed, the tool plays the inputs recorded before to operate the system and captures the screen images again. The screenshots of the first version of the system are compared to the correlating screenshots of the second version called by the same inputs. The tool displays the differences of the images and shows what components of the UI design have been changed. This tool does not evaluate the final design of the system, but demonstrates the visual differences between two versions of the same screen via image capturing. This technique could be used to compare the real application screen with the UI image created by the designer.

One of such screen capturing testing tools is Sikuli, developed by Tom Yeh, Tsung-Hsiang Chang and Robert C. Miller [4] [14] [47] [48]. It is an image recognition tool to automate the testing experience of the GUI, including a visual scripting API and an integrated development environment for writing visual scripts. Sikuli is based on the finding of target patterns on the screen and does not need access to the source code. Therefore it can be used both for desktop and mobile applications. However, mobile applications can be tested only on the desktop screen running in simulator or getting the application screen on the desktop connecting the device via VNC server. The second option can be used for Android applications without problems, but it requires the jailbreaking of the iOS device, since VNC servers are not available for iPhone/iPad.

The tool provides two core functionalities – Sikuli Script and Sikuli Search. With Sikuli Script it is possible to write visual scripts in Jython (combination of Java and Python) and to refer to UI elements using the provided library of functions and action commands. It allows the taking of a screenshot of the needed GUI component, adding it to the script and defining the action that this element should perform. The tool searches for a given component on the applications screen with a pattern matching technique, using open-source computer vision library (OpenCV). It compares the target pattern to each region on the screen of the same size, trying to find the most similar one, and is suitable for small patterns, such as buttons or icons. Sikuli also has an algorithm to detect larger patterns, like a window or dialog box using a combination of matching elements in the relation to the target pattern. Applying grayscale or multiple scales to small elements, Sikuli is able to identify color change and resized images to detect possible changes in screen resolution. The system also provides the possibility to find text elements using optical character recognition (OCR). Sikuli Search is a part of the system that enables the search of information about the selected UI object in the online documentation. It contains three components: a screenshot engine, a UI for querying the search engine and a UI for adding screenshots.

A similar visual-based black-box-driven testing tool, called eggplant [8] [12] [39] was developed by the company TestPlant. eggPlant can be used for the automation of GUI testing of desktop and mobile applications, independent of the operating system. In addition, the tool provides the image collection of the platform-based differences of GUI elements, and so supports cross-platform testing. The core principle of eggPlant is the comparing of UI image with an expected image to see to what extent they are similar. Like Sikuli, it also uses pattern matching technology to find the target images, however it runs on all platforms, including non-jailbroken iOS devices. eggPlant system consists of two parts: controller machine and the system under test, which runs a VNC server. The controller machine is used to write and execute the scripts, which can be written in the intuitive test definition language SenceTalk. eggPlant uses the image recognition algorithm

to analyze the screen, to find the selected UI component and to perform the defined action with it. The tool also supports the usage of OCR engine for text recognition and allows the testing of millions of input combinations via data-driven testing. eggPlant can be integrated in different popular test management software, such as Jenkins, IBM Rational Quality Manager and HP Quality Centre.

The advantage of the screenshot capturing method is that it considers only a visual representation of the application, and, as a result, can be used for all platforms. However, the small elements with similar color patterns can sometimes cause the image mismatch.

The research overview shows that most currently existing techniques can be divided into model-based approaches, which examine the structure of the application through finding all interactive elements and their connections, and into image recognition approaches, which capture the screen to find the required GUI components. The majority of presented tools work with the generation of automated test cases to imitate the user interaction with the application. However, these approaches for the evaluation of the GUI only analyze the functionality or the usability of the interface, including how well the users can interact with the application. None of them is taking into account the correct visual design and the representation of certain UI components and their features. Furthermore, no tools for the automated testing of the corporate design with any of these methods are known. The quality applications require not only the correct functionality, but also they also need to be visually appealing, and especially for brand applications they need to represent the corporate identity of the company. The usability of the GUI also depends on such design properties as appropriate colors, contrast and constant sizes and styles. That's why the development of the automated testing tool for the testing of the application design is of high importance.

The presented techniques can be taken as a basis for the design testing tool through automating the test experience and identifying all current elements, their attributes, such as colors, sizes, spacing and styles, and analyzing them.

3. Background

3.1. Internal structure of Volkswagen Group AppFactory

Volkswagen AG is the biggest European automotive manufacturer, with a wide variety of services. The Group Information Technology (IT) enables the security of production processes and technology, and works with innovative solutions. The Group IT contains many different departments. One of them is Group AppFactory, which is focused on mobile solutions for Volkswagen AG. It develops mobile applications for Apple, Android, BlackBerry and Windows for external but, especially, for internal customers. The products vary from the digital product catalogue and car challenge games to event applications and digital forms for employees. The goal of Group AppFactory is to optimize the existing or upcoming business processes of the company through contemporary and cost-effective deployment of mobile solutions [45]. In addition to the conception, design and development of mobile applications for customer needs, AppFactory also offers expert opinions in applications know-how and enables maintenance in permission processes, deployment and support. For the security of internal data, special mechanisms are built into mobile devices. To reduce costs, AppFactory uses standards, such as “Baukasten-System”. It allows the use of modules and re-usable components for frequent required tasks, so that they do not need to be developed again for each new application. It reduces the production costs to 70 percent.

The production of one application usually takes four to eight weeks. It starts with finding the idea, creating use cases and the ordering of the project. Next step is conception, with the preparation of requirements, development of architecture and storyboard design. After that, the realization of the application ensues. It includes the preparation of test cases and setup of infrastructure. The last step in the application production is enterprising with the application signing and deployment.

To realize this production process, AppFactory operates all tasks in different departments. So, for example the conception phase is done in the customer management part of the organization. Here the design of the application is prepared. After talking to customers, the project managers create requirements and use scenarios, also concerning the visual representation. Based on these requirements and on corporate design guidelines, the designers create the layout of the application. The design can be represented with different means. Commonly, these are images, which show every single view. The attributes of the UI elements, such as typefaces, sizes, colors and spacing, can be noted directly on the image. Sometimes the designers additionally create a list of all elements with their values in the Excel table or in the PDF file. The designers also accomplish the assets of graphical elements, such as icons and buttons, and write their names in the documents.

The developers get the design documents together with assets and implement the application based on them. When the first version of the future application is ready, it goes to the quality management department. Here the functionality, usability and design of application have to be tested. In quality management of AppFactory, the tools, such as HP QuickTest Professional and HP Application Lifecycle Management (ALM), are used for the testing purpose. The usual testing process includes creating test cases, using the above-mentioned software, running these test sets, and recording the results. Currently, the only automated part of the testing process in AppFactory is the use of test tools for the easier controlling of the process, and recording and analyzing the results. Proving the functionality

and design of the application is done by the testers manually. For the testing of design requirements of the application, the testers have to prove manually all colors, sizes and typefaces, which cannot always be defined correctly with the human eye. For that reason, the testing of design requirements is underrepresented in AppFactory and will not be done for all applications. That can lead to the case that not all wishes of customers are represented correctly. Also, the corporate identity of the company can be damaged when not all corporate design considerations are followed. Consequently, the use of an automated tool is necessary for the testing of design requirements to follow the customer and corporate considerations and to reduce the time of the testing process.

3.2. Corporate identity and corporate design

The applications developed for Volkswagen should follow the corporate design guidelines. It has an important role in representing the corporate identity of the company.

Ind described the corporate identity as an image "'formed by an organization's history, its beliefs and philosophy, the nature of its technology, its ownership, its people, the personality of its leaders, its ethical and cultural values and its strategies" [15]. It is the expression of the company's personality, behavior and culture, which distinguishes it from other organizations. Corporate design is a visual representation of corporate identity [34]. The detailed definition of corporate design is described in the paper: "Der Corporate-Design-Prozess in der Beratung am Beispiel eines neuentwickelten Simulationstools" [32]. It is the homogeneous combination of the brand design, graphic design and architecture design. It includes symbols and graphical elements, such as logo, colors, typography, and forms to reproduce visually the image and the personality of the company in different media. Corporate design contains rules, guidelines, design patterns, signs and symbols to characterize the essence of the company. These guidelines are applied to all media types to represent consistent style. Corporate design is the visual identity of the company.

A brand is the name, definition, symbol, the graphic design or the combination of these, with the purpose of identifying its products and their distinction from those of its competitors. It makes sense to differentiate between company brand and product brand. Many companies have several product brands. For example, Volkswagen AG is a company brand that includes different product or car brands, such as Volkswagen, Audi, Skoda, Porsche etc. Company brand and the product brand have two different logotypes, which are shown in the figure 4. The company and the product brands have different meanings for different stakeholder groups. So the final user is more interested in the product brand, while media, employees and shareholder are more involved in the company brands. The product brand should make the product externally recognizable; the company brand must appeal not only outside but also inside the company for all stakeholders. That's why the design and control of company brand is more complex.

The name, logo and design of the brand must be evident, promising and unique, and should symbolize the corporate identity of the company. Discrepancies in the corporate design can cause disbelief or even contempt among the public to the brand aspiration. Therefore the corporate design plays an important role in the agreement between the visual representation and the intended brand personality, and in demonstrating the current

Figure 4: Volkswagen company and product logotypes

results and upcoming goals [41]. It should support the company's reputation, which is "a perceptual representation of a company's past actions and future prospects that describes the firm's overall appeal to all its key constituents when compared to other leading rivals" [29].

That's why the following of corporate design guidelines is so important in the designing of mobile applications for Volkswagen AG. The corporate design of Volkswagen AG was developed by the design agency MetaDesign. It started to cooperate with the automotive manufacturer in 1997 with the designing of the logo, and has already realized 150 projects. With a change in legal circumstances, MetaDesign developed a new corporate design for the Volkswagen brand in 2008. The consistent and homogeneous image of both company and product brands can be found in all areas of visual communication. MetaDesign tried, through its corporate design, to reproduce the company's strategy and to place the innovation and simplicity of the brand in the foreground. So the new individual typefaces, new colors, and the new structure of representation in the web were invented [9]. The typefaces used for the company brand are Thesis and Cellini. For the digital media it is recommended to use Verdana instead of Arial. The font family VW Headline is now used for the Volkswagen product brand. Together with the logo and special developed design guidelines they give the brand the worldwide homogeneous image. To support the consist appearance of the products worldwide, MetaDesign has developed an online Volkswagen Corporate Design Net. It includes the design guidelines in several languages, as well as templates and examples.

4. Defining the Design Guidelines for Mobile Devices

Media design plays an essential role in the user decision on the product and in supporting the brand's potential. That's why it is important that mobile applications have a suitable representation that meets all user expectations. Much successful work has been done in developing manuals for the interface design of different interactive systems. Despite that fact, the design of mobile applications remains a relatively new and undocumented topic. Some of the universal principles can be applied for the mobile devices as well, but there are a lot of significant differences that need to be taken into account in designing the interface of mobile applications. These include limitations in memory, battery life, small screen sizes, ability to adapt to different sizes and orientations, security, network bandwidth and different input options. These functionality constraints and possibilities certainly have an influence on designing the user interface. Another important issue is the difference in platforms, in options they provide, and in appearance of their standard UI elements. For example, iOS design patterns define that tabbed navigation should be at the bottom; while in Android devices it should be placed at the top of the screen. Different operating systems have established their own guidelines – Mobile Human Interface Guidelines (HIG) for iOS, Android User Interface Guidelines, User Experience Design Guidelines for Windows Phone, UI Guidelines for BlackBerry Smartphones and PlayBook and others. Currently, there is no universal document that describes design requirements and considerations valid for all mobile devices and applications.

Moreover, most existing guidelines describe the usability and general design requirements, such as structure of the application, grouping of elements, intuitive layout, giving feedback, amount of shown information and usage of navigation controls and search functions, but they do not explicitly describe the visual appearance, e.g. recommended colors and sizes of the elements. These requirements can be effortlessly tested by an individual; however, for the machine, it is challenging to decide if the interface is nice-looking when no concrete values are given. There have already been some usability testing tools presented in the recent years, but there is a lack in considerations concerning the actual visual characteristics, making the application appealing and consistent, so that the user can associate the graphical components with their functionality.

The consistence of the design is also important for the corporate identity in order to represent the image and distinctiveness of the company. Corporate design contains the rules, guidelines, and design patterns for the usage of typical signs and symbols in different media types to create a homogeneous style and to reproduce the company identity. The corporate design of Volkswagen AG includes the logotype, house fonts, house colors etc.

For the automated design-testing tool it is important to note that not every application has the same requirements. The customers decide if the applications should follow the corporate design or if they prefer their own one. Some of the general considerations can be applicable for all applications. For example, buttons must be large enough for the user to be able to tap them. Background and text must have contrasting colors to enable the user to read the text without problems. In addition, the same elements must have the same color, since the researches state that we often use color for grouping or connecting things to each other. But other considerations depend on customer needs, as well as on the

operating system. That’s why a choice should be made between company, product or custom requirements before starting the test. I have created a list of design requirements that should be proved in the automated testing tool, based on the following sources: “Designing the Use Interface: Strategies for Effective Human-Computer Interaction” by Ben Shneiderman [31], “The Elements of User Interface Design” by Theo Mandel [21], “Guidelines for Enterprise-Wide GUI Design” by Susan Weinschenk and Sarah C. Yeo [46], iOS Human Interface Guidelines [17] (as I am concentrating on the iOS applications in my work), VW Company Corporate Design Guideline [43], VW Product Corporate Design Guideline [44], as well as an interview with the designers Martin Bonneberg and Jennifer Jane Poerner. The general and corporate design requirements are presented in the tables 1 and 2. Concerning the customer design requirements, the Excel template was created for entering the attributes. All custom attributes of given UI components in the table (font family, font size, font color, background color, alignment, width, height, margins) must correspond with the same attributes in the application.

General design requirements	
1	Consistency must be followed throughout the whole application: - All buttons must have the same height - Buttons must have not more than 2 different widths - If buttons or icons are placed in one line, they must have the same margin from each otherand from the screen sides - All icons must have the same size - All labels of the same type (titles, subtitles, button text) must have the same font family, font size and font color - Font family must be consistent in the whole application, if necessary one alternative font family can be used
2	Color contrast between background color and font color must be at least 50%, font color must be darker than background color
3	Interactive elements (links) must have another color than non-interactive elements (normal text)
4	The number of different colors in the application must be from 4 to 7
5	Normal text must have regular or medium weight, not bold or light
6	Radio buttons can be used if the number of options is less than 6. Check buttons can be used if the number of options is less than 10. If the number of options is 10 and more list boxes must be used
7	All buttons, toolbar and navigation bar icons and other tappable controls must have the minimum size of 44x44 px (iPad2 – 22x22 px)
8	Tab bar icon must have the minimum size of 50x50 px (iPad2 – 25x25 px)
9	Launch image must have the size of: 640x1136 px (iPhone5) 640x960p px (iPhone) 1536x2048 px (iPad and iPad mini) 768x1024 px (iPad2)

Table 1: General design requirements

Volkswagen corporate design requirements		
	Company CD	**Product CD**
Content		
1	Font Family: Thesis TheSans, FF Cellini or Verdana	Font Family: VW Headline OT, VW Utopia or Arial
2		Title Font Family: VW Headline OT
3		Title Font Color: Black (0/0/0)
4	Font Color: VWAG Grey (76/83/86)	Font Color: Pantone 432 (51/67/76)
5	Font Size: 15 px	Font Size: 18px
6	Content Background Color: White (255/255/255)	Content Background Color: White (255/255/255)
Headline		
7	Headline Font Color: VWAG Grey (76/83/86)	Headline Font Color: Pantone 432 (51/67/76)
8	Headline Font Size: 32px	Headline Font Size: 32px
9	Subheadline Font Size: 15px	
10	Headline Background Color: White (255/255/255)	Header Background Color: White (255/255/255)
Buttons		
11	Button Text Color: VWAG Grey (76/83/86), Grey (79/84/89) or White (255/255/255)	Button Text Color: White (255/255/255) or Dark Grey (17/17/17)
12	Button Text Font Size: 14px	Button Text Font Size: 15 px
13	Button vertical Gradient: FU1: 219/219/220, Pos.: 0 %, FU2: 241/241/242, Pos.: 50%, FU3: 255/255/255, Pos.: 60% Pushed: FU1: 159/163/169, Pos.: 0%, FU2: 173/178/184, Pos.: 50%, FU3: 187/192/199, Pos.: 60%	Button Interactive Blue Gradient: FU1: 34/116/172, Pos.: 50 %, FU2: 99/157/197, Pos.: 95% Button Interactive Orange Gradient: FU1: 255/135/31, Pos.:50%, FU2: 255/184/121, Pos.: 95%
14		Button Grey and Button Inactive Gradient: FU1: 186/194/197, Pos.: 50%, FU2: 234/238/237, Pos.: 95%
15	Delete-Button Color: FU1: 188/26/34, Pos.: 0%, FU2: 220/23/54, Pos.: 50%, FU3: 234/0/45, Pos.: 60%, FU4: 236/0/46, Pos.: 100% Pushed: FU1: 145/20/27, Pos.: 0%, FU2: 170/24/31, Pos.: 50%, FU3: 185/26/34, Pos.: 60%, FU4: 188/26/34, Pos.: 100%	
16	Delete-Button Font Color: White (255/255/255)	
TableView / ListView		
17	Table Background Color Left Column: VWAG Petrol light 50% (231/243/243)	Table Background Color Left Column: Pantone 427 (234/238/237)
18	Table Left Column Width: 320 px	
19	Table Left Column Cell Height: 44 px	

20	Table Font Color: VWAG Grey (76/83/86)	Table Font Color: Pantone 432 (51/67/76)
21	Table Font Size: 15px	Table Font Size: 18px
22	Table Header Font Color:VWAG Petrol 60% (103/135/157)	Table Header Font Color: Pantone 430 (137/148/160)
Borderline		
23	Borderline Color: VWAG Silver 40% (228/228/29)	Borderline Color: Pantone 427 (234/238/237)
Count Indicator		
24	Count Indicator Color: VWAG Petrol 60% (103/135/157)	Count Indicator Color: Pantone 430 (137/148/160)
25	Count Indicator Font Size: 15px	Count Indicator Font Size: 18px
Main menu slider		
26	Main menu Slider Height: 192 px (opened), 52 px (closed)	Main menu Slider Height: 192 px (opened), 52 px (closed)
27	Main menu Slider Background Color: VWAG Petrol (0/70/102)	Main menu Slider Background Color: Pantone 432 (51/67/76)
28	Main menu Slider Label Color: VWAG Petrol light (198/223/231)	Main menu Slider Label Color: Pantone 299 (0/177/235)
29	Main menu Slider Status Color: Red (224/8/8) or Green (34/210/47)	Main menu Slider Status Color: Pantone 390 (185/201/0) or Pantone 186 (228/0/44)
30	Main menu Slider Header Font Size:18px	Main menu Slider Header Font Size: 22 px
31	Main menu Slider Labels Font size:15px	Main menu Slider Labels Font size: 18 px
Vertical slider		
32	Vertical Slider Width: 320 px	Vertical Slider Width: 320 px
33	Vertical Slider Cell Height: 44 px	Vertical Slider Cell Height: 44 px
34	Vertical Slider Font Size: 11 px	Vertical Slider Font Size: 11 px
Navigation bar		
35	Navigation Bar height: 45 px	
36	Navigation Bar Title Font Size: 18 px	Navigation Bar Title Font Size: 22 px
37	Navigation Bar Search Font Size: 15 px	Navigation Bar Search Font Size: 18 px
Icons		
38	Icon Size: 48x48 px (main menu) and 28x28 px (submenu)	Icon Size: 48x48 px (main menu) and 28x28 px (submenu)
Logotype		
39	Logotype Alignment: centered	Logotype Alignment: right
40	Logotype Margin left/right: minimum half ofthe Logotype Width	Logotype Margin right: 15 px (iPad and iPhone3), 30 px (iPhone4)
41		Logotype Margin left: Logotype Width
42		Logotype Margin top: 7 px (iPad and iPhone3), 15 px (iPhone4)
43	Logotype Width: 180 px (Login-Screen) and 120 px (Content)	Logotype Size: 30x30 px (iPad and iPhone3), 60x60 px (iPhone4)
44	Logotype Background Color: White (255/255/255)	
Login-Screen		
45	Login-Screen Background Color: VWAG Silver 40% (228/228/29)	

46	Login-Screen Header Color: VWAG Petrol (0/70/102)	
47	Login-Button Font Color: White (255/255/255)	
48	Login-Button Font Size: 14px	
49	Login-Button Color: VWAG Silver (168/173/179)or FU1: 148/153/158, Pos.: 0%,FU2: 167/172/178, Pos.: 50%,FU3: 179/185/191, Pos.: 60%,FU4: 190/195/202, Pos.: 10 % Pushed: VWAG Red (162/30/77) or FU1: 160/50/88, Pos.: 0%,FU2: 190/89/122, Pos.: 50%,FU3: 194/109/136, Pos.: 60%,FU4: 194/109/136, Pos.: 100%	

Table 2: Volkswagen corporate design requirements

5. The DesignTesting Framework: Approach for the Automated Design Testing Tool

5.1. Scientific question

Current work done in the area of mobile application testing shows that there are still leaks in effective automated testing of UI design, considering the visual representation of the GUI elements. In particular, there are no tools existing on the market for examining the company-specific requirements and corporate design of brand applications. The development of such an automated tool could improve the effectiveness of the testing process and save time and costs. The goal of this master thesis is to develop a prototype for the automated testing of design guidelines, including the given color schemes, spacing, sizes and corporate identity of the company, and to evaluate how this method influences the testing experience. The scientific question of this master thesis is, therefore, the following: With what method can the specified design requirements be automatically tested with different versions of mobile applications?

The target group of this system is the quality management department of application developing companies, in this case of Volkswagen Group AppFactory. This means that the users of the automated design testing tool are people of working age, who have already experience with mobile technology and with a testing process of mobile applications.

5.2. Different ideas

As the research overview has shown, there are two core possibilities for the solution of this problem – a source code analysis of the applications structure and a screen capture analysis. So I developed different ideas based on these techniques to review their advantages and disadvantages.

5.2.1. Source code analysis of layout files

One possibility to test the design of the mobile application is the analysis of layout files, for example XML files for Android, XIB files for iOS, and XAML files for Windows 7. The layout files of most operating systems are XML-based and have similar tree structure containing the hierarchy of UI elements, so it is possible to develop a similar approach for all platforms. Nevertheless, it should be implemented differently for each type of layout file, since they still have some differences.

The core idea is that the program reads the layout file, goes recursively through all parent and child nodes, searches for required nodes with a certain type, and returns their attributes, which are also implemented in the layout file. Finally, found UI elements can be compared with the given design requirements. As an example of this approach, I have implemented a java program that evaluates the XML files of Android applications (Figure 5).

```java
public static void readXML() throws Exception{
    try {
            File fXmlFile = new File("src/app_sample.xml");
            DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
            Document doc = dBuilder.parse(fXmlFile);

            doc.getDocumentElement().normalize();

            //String expression;
            NodeList buttonList = doc.getElementsByTagName("Button");
            NodeList textList = doc.getElementsByTagName("TextView");
            NodeList iconList = doc.getElementsByTagName("ImageView");

            //Icon Comparison
            for (int i = 0; i <iconList.getLength(); i++){
                Element eIcon = (org.w3c.dom.Element) iconList.item(i);

                if (eIcon.hasAttribute("android:id") && eIcon.getAttribute("android:id").

                    iconMarginTopXML = eIcon.getAttribute("android:layout_marginTop");
                    iconMarginRightXML = eIcon.getAttribute("android:layout_marginRight")
                }
            }

            // Text Comparison
            for (int j = 0; j <textList.getLength(); j++){
                Element eText = (org.w3c.dom.Element) textList.item(j);

                if (eText.hasAttribute("android:text") && eText.getAttribute("android:te

                    titleTypefaceXML = eText.getAttribute("android:typeface");
                }
            }

            // Buttons Comparison

            for (int temp = 0; temp < buttonList.getLength(); temp++) {

                Element eButton = (org.w3c.dom.Element) buttonList.item(temp);

                // Search Button
                if (eButton.hasAttribute("android:id") && eButton.getAttribute("android:i

                    buttonSearchTextcolorXML = eButton.getAttribute("android:textColor");
                    buttonSearchColorXML = eButton.getAttribute("android:color");
                }

                // Cancel Button
                if (eButton.hasAttribute("android:text") && eButton.getAttribute("androic

                    buttonCancelTextcolorXML = eButton.getAttribute("android:textColor");
                    buttonCancelColorXML = eButton.getAttribute("android:color");
                }
            }

            } catch (Exception e) {
            e.printStackTrace();
            }

}
```

Figure 5: Java implementation of the layout files source code analysis approach

The weakness of this method is that usually not all of the UI components and their features are implemented in the responsible layout file. Especially in Android applications, there is a separation between layout and the appearance of UI elements. Some of the layout

information can be set not only in the layout files of every view but also programmatically in the whole source code. The appearance of UI elements is often specified in the theme files and styles that are applicable for the whole application. The design of iOS applications is often defined using an integrated Interface Builder. This can be done almost without the use of code. After debugging, this layout information is saved in NIB files, which are the not editable versions of XIB files.

In addition, some attributes can be dynamically changing during the runtime of the application. Although all XML-based languages have the tree structure, the names of the nodes are different. For example, the screen node in Android is called Activity, in iOS it is UIViewController, and in Windows 7 applications it is PhoneApplicationPage. The list items in Android and Windows 7 are presented in ListView and ListBox correspondingly, while in iOS they are stored in UITableView. Moreover, the layout files do not have information about the connection between different screens: these relations are described in the source code. Consequently, to have full and correct knowledge about the UI elements it is not enough to analyze only the layout files responsible for the each view, but it is also important to take into account the whole source code.

5.2.2. Source code analysis of the application code

Another approach includes the reconstruction of the whole application and its resources. It analyzes not only the layout files but also the whole source code through searching for the relations between different views and UI elements. There are various tools and techniques presented in the related work, both for Android and iOS, that are able to extract the called functions and UI elements and represent them and their values in the hierarchical structure.

While in Android applications most connections between the screens are described in the manifest file, in iOS applications the relations are stored in different ViewContollers through the whole application. So this technique must be implemented independently for each operating system because of the significant differences in their architecture.

5.2.3. Screenshot analysis through image recognition tools

The existing screen capturing tools, such as Sikuli and eggplant, make it possible to automate the process of design testing using custom scripts. These tools are able to define the color, size, position of detected images, and even the font family of the text, using OCR. But unlike the source code analysis, the image recognition tool cannot refer to certain UI elements using their ids. That means that the tool should be able to identify UI components according to their visual appearance, and all UI elements that need to be tested, should be available as images. Then the tool can search these assets on the screen and define the required attributes according to the type of the element. The information about each component can be read from the design requirement file. The advantage of this method is the platform independence; it can be used with any operating system. Also, there is no problem with the relation of views and UI elements, because the tool can find only those components which are located on the current screen. But the weakness of this approach is that it considers the application screen only as an image and cannot recognize the types of all UI elements (labels, buttons, icons etc.) and relate them

to the corresponding components from the design requirements document. In addition, the search for the label element can be difficult because its text can be dynamically changed due to the user input. Moreover, the image analysis algorithm is not as accurate as the analyzing of source code information, especially by small elements mistakes can occur.

5.2.4. Screenshot analysis through image comparison

Design of the application includes not only the listing of all elements and their values but also representing them at the image. So the designers commonly provide images of all states of the application, which illustrate how the screens should look. A possible opportunity of the design analysis is the total image comparison of the illustration created by designer and the screen capture of the application view. Two images can be compared using data comparison, pixel-to-pixel comparison, or feature-based comparison, searching for the positions where two images are different. These differences can be highlighted using the color difference technique. To define what UI elements are affected, it is essential to use an additional technique, such as identifying the element by the means of image recognition tool or through the source code analysis knowing the coordinates of the object. In this case, it is not necessary to examine all components, but only those which differ. However, this technique can evaluate only static UI elements, because every dynamical change will be considered as a failure.

5.2.5. Combination of various methods

As can be seen, each design testing idea has its weaknesses. The source code analysis can return more accurate results, but the elements described in the code can be rendered differently on the screen. The image capture analysis allows the testing of only that information which can be really seen in the application. So the combination of both source code analysis and image recognition techniques can return better results.

The combined approach can use the image recognition to find all UI elements visible on the screen at the current time and identify them by comparing with all available assets. Another possibility to find all visible elements is to search for them in the source code, save the images of every element and compare them with screen capture of the application. Then different kinds of attributes can be defined using source code analysis or image recognition tools. For example, the static information about colors and font families can be easily extracted from the source code, while the dynamic spacing constraints between certain elements can be better found with image recognition, so the source code does not need to search for the closest components.

The limitation of this method is that not all assets of the UI components can be available before testing, and some components defined in the source code cannot be correlated with their visual representation on the screen, if they include the dynamically changing text element. In addition, this technique is time-consuming. The cases when the GUI is rendered differently as described in the code are not very common. That’s why the use of only one technique will bring sufficient results.

I decided to implement an automated design-testing tool, based on the dynamic source code analysis technique, since it can be executed faster and returns more accurate results.

It also allows one to easily identify and relate the UI elements defined in the design requirements and in the application.

5.3. Approach for automated design testing tool

5.3.1. General idea

The DesignTesting Framework is an automated tool for the iOS applications that can read the structure of the UI elements and define their attributes. It can be used within the XCode project of a mobile application. My framework was implemented in Objective-C and can be linked to the target of any iOS application with the available source code information. Since the system is developed for the use in companies for testing their corporate design, the source code information should be commonly available.

The DesignTesting Framework can be linked to any iOS application with the available code. So it will automatically be debugged when running the application and can be activated through the shaking movement of the device. When the user shakes the device with the running application, the DesignTesting Framework is executed. It goes recursively through the source code of the current view, reads the application structure and searches for all present subview elements. It determines the id number (tag) and the type of every subview visible on the screen (label, button, image, table etc.). Then the system reads the excel documents in the comma separated CSV format with the customer and corporate design requirements and compares this data with the attributes of the elements in the application, according to their ids. An output of the testing is a PDF document containing information about right and wrong attributes of all elements visible on the screen. The flow of the testing process with my automated tool is demonstrated in the figure 6.

In order to use this framework in the testing phase, the design and implementation of the application must follow some restrictions. The designers and programmers must provide every UI component with the unique id number that must be the same in the design documentation and in the source code. The designer must fill out the excel template with the ids, element names and all needed attributes of the elements. This information will be used for the testing of customer design requirements of the application. The testing of general and corporate design requirements does not need to fill out the excel table every time, because this information commonly does not change.

This method can create the list of all available UI components very quickly and determine the precise values that are not able to be defined so accurately with the human eye or the image recognition technique. It can correlate the UI elements to the corresponding assets in the table without problems, using ids.

5.3.1. Physical constraints

The DesignTesting Framework was developed in Objective-C using XCode. For the testing of iOS applications, it requires the computer with OSX operating system with installed XCode and iTunes. Since the system is storing data in iTunes connected with the mobile device, it cannot be used with the simulator. Therefore the tool also needs an iOS mobile device (iPad or iPhone) to run an application on it.

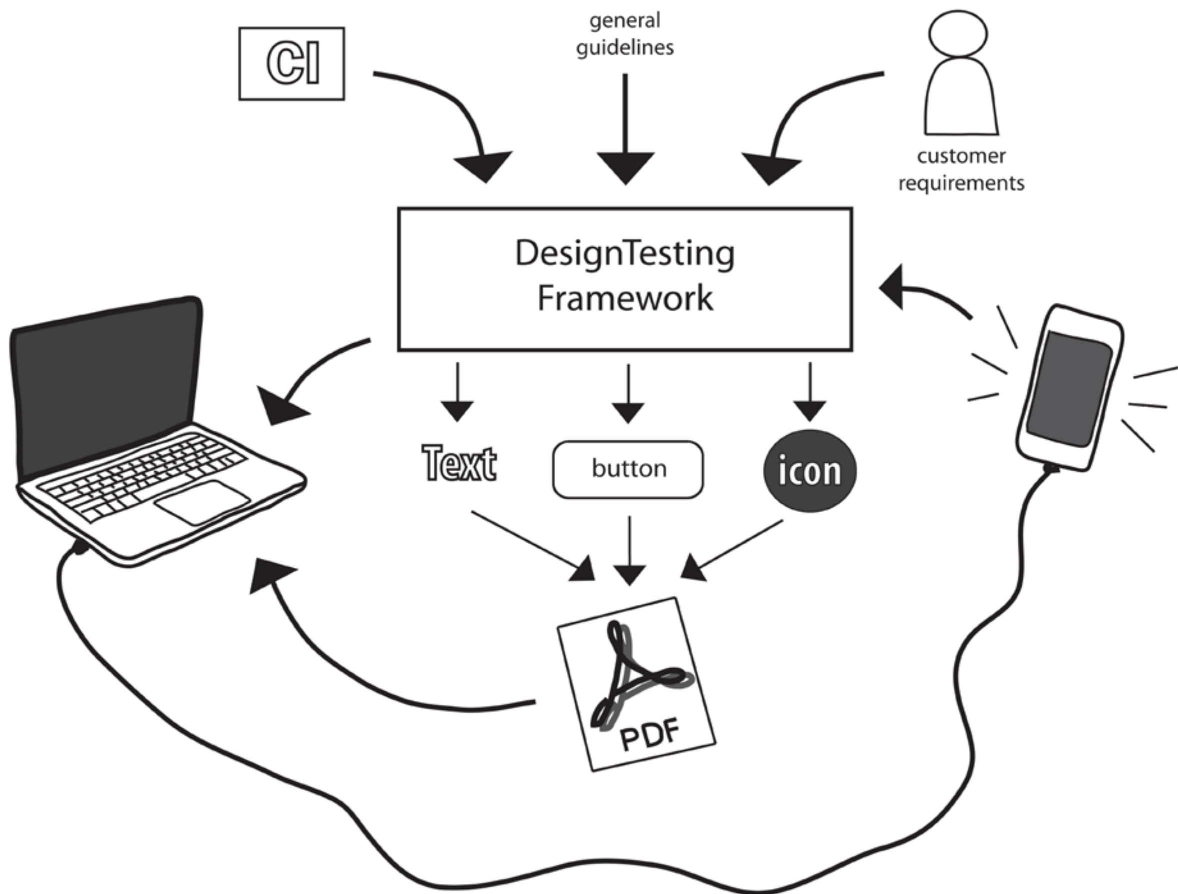

Figure 6: Structure of the automated design testing tool

5.3.2. Functional overview

To offer the easy and effective design testing experience, my DesignTesting Framework provides the following functional and non-functional specifications:

- Working with any iOS application with the available source code

The system source code is bundled in a framework, so it can be copied to the framework library and linked in the target's Link Binary with Libraries - Build Phases section of any application within the XCode project. Once built, the application can activate the design testing functionality at any time through the shaking gesture. If the design testing functionality is not needed anymore, the framework can be removed from the XCode project before releasing the application.

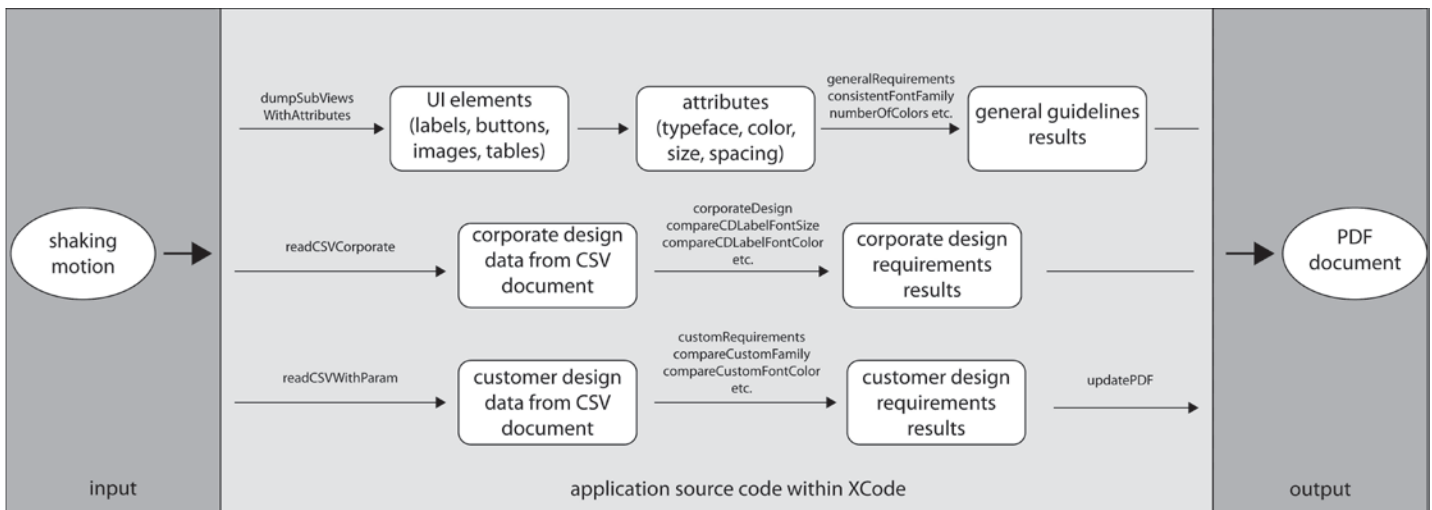

Figure 7: Data flow of DesignTesting Framework

- Orientation and resolution awareness

The mobile applications can be developed for only one or for both portrait and landscape mode. If the application is using both orientations, the design must be adjusted for both of them and described in two excel documents for portrait and landscape modes accordingly. The framework defines the current orientation of the device and loads the corresponding excel document. Many design requirements have two different sets of assets and attribute values, dependent on whether the device has a low or high resolution. The excel table contains the size and spacing information for both low and high-resolution devices. The framework defines the resolution of the device under test and reads only those values from the table that are associated with the low or high-resolution screens.

- Setting of user preferences

For the easier evaluation of the design testing results, the system provides the possibility to set the user preferences in the device settings. Through the Settings.bundle included in the DesignTesting Framework, it is possible to choose if the application under test is a company or product application, if the tester wants to prove the customer or corporate design requirements, and if the output document should contain only wrong, only right, or all results. All these user preferences can be set in the settings section of the application (Figure 8).

- Storing data with iTunes

The resources used for the testing are stored in iTunes, which automatically synchronizes the using documents between computer with the XCode project and the application on the mobile device. All documents required for the test (Excel files, assets) must be loaded in the application section in iTunes while the device is connected to the computer. Even if the device is not connected to the computer during testing, the framework is able to read the resource files from iTunes. When the test is executed the results are saved in iTunes and can be then copied to the computer for evaluation (Figure 9).

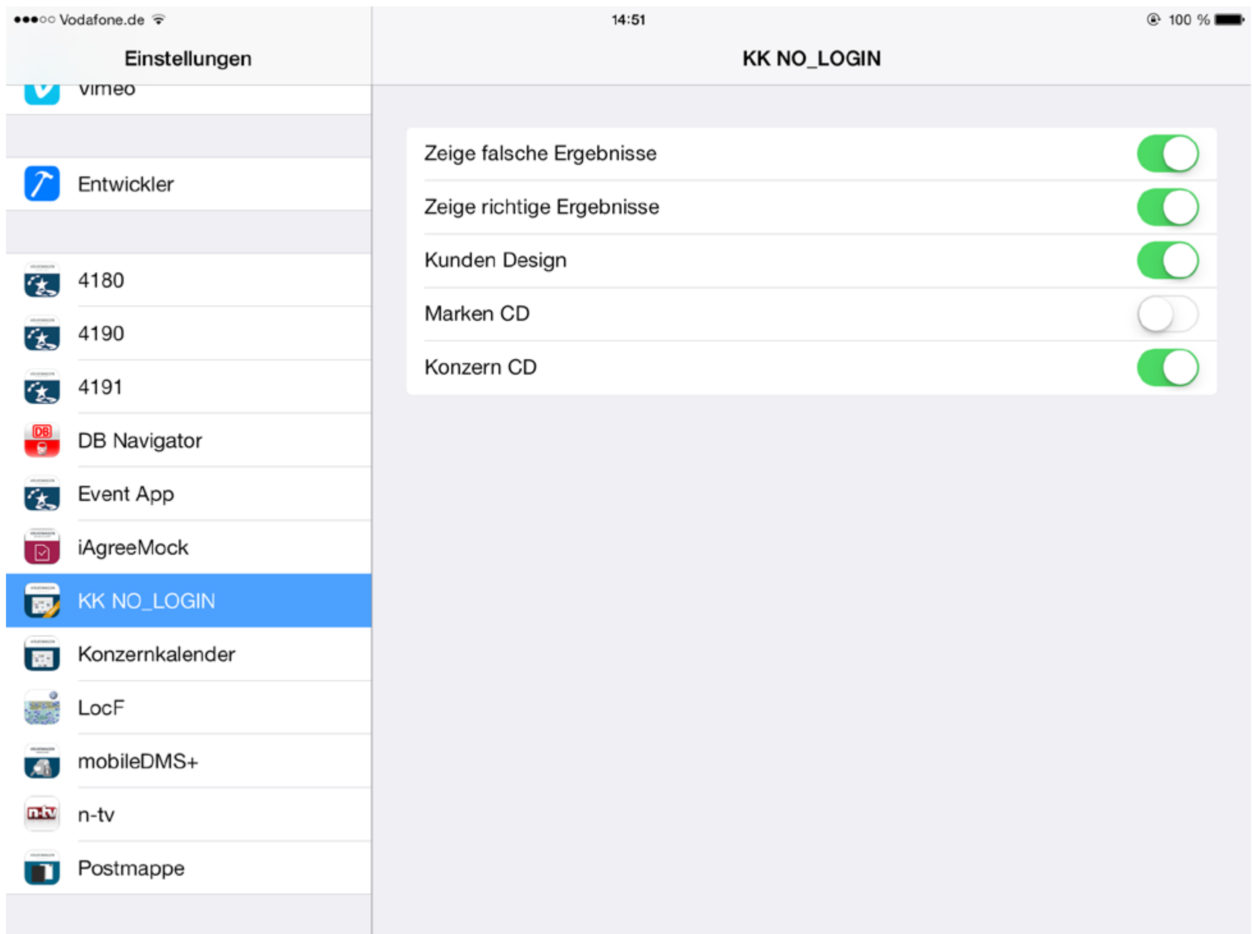

Figure 8: Screenshot of user preferences settings

Figure 9: Screenshot of storing data with iTunes function

- Finding all UI elements of the current view

In Objective-C, every UI component is represented as a subview. All subviews are the nodes of the hierarchical tree structure, where the highest node is a screen view. Objective-C provides a possibility to list the subviews currently visible on the screen. The system goes recursively through the array of all subviews and their childs, searching for all UI elements in the tree structure. All subviews belong to a certain type. The system checks the type and considers only those element types that are essential for design testing, such as labels (UILabel), buttons (UIButton), images (UIImageView) and tables (UITableView).

- Defining the certain attributes of required elements

For every UI element, the system determines the attribute values. Some values can be established only for a certain type of the element. So the images do not have any font information. For the labels it is necessary to know the font family, font size, font color and alignment. For the buttons, in addition to the title font information, the system defines the background color. The width, height and spacing from the sides can be defined for all elements. All numbers are converted into decimal integer values and all colors are converted in RGB values.

- Reading the design requirements from the excel document

In order to compare the design requirements with the application values, it is necessary to load the data in an appropriate format. The customer design requirements should be filled out in the Excel template. The document is then converted in a comma-separated CVS format with the help of macro script. This format can be easily read and interpreted with the Objective-C. The corporate design requirements are also stored in the CVS file, but since theyare usually not changing, it is not necessary to fill out the values and convert the document for every new application. The general requirements are described in the source code, since they are the same for all applications.

- Comparing UI element attributes with the customer and corporate design requirements

The attribute values found in the application source can be compared with the values stored in the Excel table. All UI elements (at least those which are essential for the design testing) in the table and in the application have their unique id number. The elements from the application and from the design requirements table can be associated with each other with the help of ids. The system compares the defined attributes and returns the output, whether the values are right or wrong. If the table contains no values for a certain element, it means that these values do not have any importance for the visual design and will be not compared.

- Analyzing the general design requirements

The system also proves the general requirements of the application, such as consistent font families, font sizes and button heights, number of colors and a contrast between button background color and title color. It saves all fonts, sizes and colors that have been tested in an array and examines at the end how many different font families, sizes and colors the array contains. The color contrast is calculated through the difference between font luminance and background luminance and should be at least 50 percent.

- Comparing the screenshots of the UI elements with the assets

The system takes an image capture of all UI elements that have been tested and saves them in a folder. This way the UI elements can be compared with the assets defined in the design requirements table. It is an important step, because some buttons and text elements are represented as images, so that they do not have any font or background color information. In this case, the system only compares whether they correspond to the assets associated with them. For the images, it verifies if the correct asset was used for them. The scaled UI elements are resized to the size of the original asset. For better results, the image comparison uses two techniques. First, the images are compared using data comparison. This technique is very fast and has an exact result, but it works only for the images which are totally similar and have not been scaled. If the image is right, it does not need any further comparison. If the image is not right it is analyzed again with the help of pixel-by-pixel comparison. This method is slower but can calculate the difference percentage of two images. If the difference is less than 10 percent, the images can be regarded as similar. However, it depends on the total number of pixels. For large images this difference can be bigger, and for the small icons the difference is smaller. Also, in the case of very similar images (for example white background with two different black texts), it can happen that the pixel-by-pixel comparison calculates a very small difference. So if the difference percentage is more than 5 percent, it is recommended to perform an additional comparison by someone else.

- Result output

All testing results, including general consideration, customer and corporate design requirements, are rendered in a PDF file, which is saved in iTunes. Depending on the user settings preferences, the output file contains wrong and/or right information about customer and/or corporate design of all UI elements, and information about the general design of the whole application screen. For better visualization, the right results are highlighted in green and wrong results in red.

5.3.3. Architecture

The architecture of the DesignTesting Framework is presented in ULM diagram in figure 15. The system consists of 2 classes – DesignTestingViewController and VWCVSAttributes, and uses the classes PDFConverter and PDRRenderer, developed by SorinNistor [37] for converting the PDF files to PNG images. DesignTestingViewController is called when the application starts. It includes 3 core methods, which are essential for the application functionality:

- motionEnded:(UIEventSubtype)motion withEvent:(UIEvent*)event

This method is continuously checking for the shaking gesture and calls the main method when the shaking motion is ended.

- main

Main method is executed when the user shakes the device. It contains the information about user settings, defines the resources directory, calls a DumpSubview method and creates an output PDF file.

- dumpSubViewsWithAttributes:(VWGCVSAttributes *)cvsAttributes

DumpSubviews is the core functionality of the framework. It goes recursively through all subviews of the current application screen and creates the list of all UI elements (Figure 10). For each type of found UI elements (currently for labels, images, buttons, tables and table cells), it requests the information about the attributes of the elements.

VWCVSAttributes class contains all property variables and helping methods for reading the Excel document, defining the attributes and comparing them with the design requirements.

- detectOrientation

This method detects the orientation of the application in order to load an appropriate design requirement document. It is necessary to mention that it does not detect the device orientation, because some applications were developed only for one mode.

- detectDeviceType

To be aware of screen proportions and other necessary settings, the system detects the type of testing device (iPhone or iPad).

- detectScreenResolution

Also the screen resolution plays an important role for the testing of design requirements. Typically, the proper assets and attributes are automatically loaded by the application. This method detects the screen resolution of the testing device to load the correct design requirements associated with them.

- checkUserSettings

The system provides the possibility to set the user preferences: if the framework should test customer or corporate design and output right, false, or both results. This method checks what preferences were set by the user.

- generalRequirements

This method contains a list of general requirements that always have to be tested, independent of user settings. It includes consistency of the application, minimum button height and number of different colors. It calls the following methods to check these requirements.

```
247 - (void) dumpSubViewsWithAttributes:(VWGCVSAttributes *)cvsAttributes{
248
249     NSArray *subviews = self.subviews;
250     UIWindow *window = [UIApplication sharedApplication].keyWindow;
251     UIView *subview;
252
253     for (subview in subviews) {
254
255         int tag = cvsAttributes.elementTag;
256         tag = subview.tag;
257
258         if (tag != 0 && !([cvsAttributes.tagArray containsObject:[NSNumber numberWithInt:tag]])){
259
260             [cvsAttributes.output appendString: @"__________________________________________________ \n"];
261             NSLog(@"tag: %d", tag);
262
263             cvsAttributes.width = subview.frame.size.width; // Width
264             cvsAttributes.height = subview.frame.size.height; // Height
265

            <...>

299             [cvsAttributes readCSVWithParam:tag];
300
301             [cvsAttributes compareImagesForSubview:subview tag:tag tagArray:cvsAttributes.tagArray];
302
303             }
304
305             // Label
306             if (tag != 0 && !([cvsAttributes.tagArray containsObject:[NSNumber numberWithInt:tag]])){
307                 if ([subview isKindOfClass: [UILabel class]]) {
308                     [cvsAttributes dumpLabelsForSubview:subview output:cvsAttributes.output name:cvsAttributes.csvName];
309                 }
310             }
311
312             //Image
313             if (tag != 0){
314                 if ([subview isKindOfClass: [UIImageView class]]) {
315                     [cvsAttributes dumpImagesForSubview:subview tag:tag tagArray:cvsAttributes.tagArray];
316                 }
317             }
318
319             //Button
320             if (tag != 0){
321                 if ([subview isKindOfClass: [UIButton class]]) {
322                     [cvsAttributes dumpButtonsForSubview:subview tag:tag tagArray:cvsAttributes.tagArray];
323                 }
324             }
325
326             // TableView
327             //if (tag != 0){
328             if ([subview isKindOfClass: [UITableView class]]) {
329                 [cvsAttributes dumpTableViewForSubview:subview tag:tag tagArray:cvsAttributes.tagArray];
330             }
```

Figure 10: Implementation of dumpSubviews function

- consistentFontFamily, consistentButtonFont, consistentButtonHeight

All font families, font sizes and button heights of the elements in the application must be consistent. While going through all labels and buttons, these methods add their font families, sizes and button heights in the arrays, and finally check whether all elements in the array are the same. For this, they go through the arrays and compare the n and n+1 members of the array. If at least one member of the array is different, the application is not consistent (Figure 11).

```
- (void)checkConsistency{
    [self consistentFontFamily];
    [self consistentButtonFont];
    [self consistentButtonHeight];
}

- (void)consistentFontFamily{

    NSString *schriftartGleich = TEXT(@"SchriftartGleich");
    NSString *schriftartNichtGleich = TEXT(@"SchriftartNichtGleich");
    NSMutableString *resultText = [[NSMutableString alloc] init];
    [resultText appendString: [NSString stringWithFormat:@"%@", schriftartGleich]];

    BOOL consistentFontFamily = true;
    if ([self.textArray count] > 1){
        int best = 1;
        for (int temp = 0; temp < [self.textArray count]; temp++){
               NSString *a = [self.textArray objectAtIndex:best];
               NSString *b = [self.textArray objectAtIndex:temp];
               if ([a isEqualToString:b]){
                   best = temp;
               }
               else {
                   best = temp;
                   [resultText replaceOccurrencesOfString:[NSString stringWithFormat:@"%@",schriftartGleich]
                                              withString:[NSString stringWithFormat:@"%@",schriftartNichtGleich]
                                                 options:NSLiteralSearch
                                                   range:NSMakeRange(0, resultText.length)];
                   consistentFontFamily = false;
                   self.consistency = false;
               }
        }
    }
    if(consistentFontFamily == true && self.displayTrue == true){
        [self.output2 appendString: [NSString stringWithFormat: @"%@ \n", resultText]];
    }
    if(consistentFontFamily == false && self.displayFalse == true){
        [self.output2 appendString: [NSString stringWithFormat: @"%@ \n", resultText]];
    }
    NSLog(@"Schriftart: %@", resultText);
    }
```

Figure 11: Implementation of check consistency function

- minimumButtonHeight

According to the design guidelines for mobile devices, the buttons should have the minimum height of 44 px, so that the user can easily tap on them. This method adds all button elements with the height less than 44 px in the array. If the length of this array is more than zero, then at least one button has the height less than 44 px and the buttons height is not correct.

- numberOfColors

The single view of the application should not contain too many different colors. Many designers suggest that it should have no more than 7 colors. While going through all elements, this method adds all colors in the array. This it checks if the color already exists in the array and adds only new ones. If the length of the array is more than 7, the application view contains more than 7 different colors.

- calculateColorDifference

The contrast between the button background and title should be sufficient to be able to read the text. To calculate the color difference, it is necessary to calculate the luminance of the background color and font color and to subtract them. The color luminance is calculated using the formula:

Luminance=sqrt((0.2126*pow(red,2)+0.7152*pow(green,2)+0.0722* pow(blue,2)))*100;
Luminance=sqrt(0.2126*red^2+0.7152*green^2+0.0722* blue^2)*100;

- corporateDesignBackgroundWithColor:(NSString *)colorStringbackground

This method detects and checks the background color of the main screen, which should be white according to the corporate design requirements.

- corporateDesign

This method contains the list of requirements that must be tested. It reads the CSV file with the corporate design requirements and calls the following methods to test the defined requirements.

- compareCDHeadline,
  compareCDLabelFontFamilyWithOutput:(NSMutableString *)output,
  compareCDLabelFontColor,
  compareCDLabelFontSize,
  compareCDButtonFontFamily,
  compareCDButtonFontColor,
  compareCDButtonFontsize,
  compareCDTableLeftColumnColor,
  compareCDTableLeftColumnWidth,
  compareCDTableFontColor,
  compareCDTableFontSize,
  compareCDLogotypeForSubview:(UIView *)subview,
  compareLogotypeGroupMarginsForSubview:(UIView *)subview,
  compareLogotypeGroupSize,
  compareLogotypeGroupBackground,
  compareLogotypeBrandMargins,
  compareLogotypeBrandSize

All these methods get the required attributes of UI elements and compare them with the values in the CSV file, according to the corporate design requirements. The connection between application and CVS objects is dealt with through the objects ids.

- dumpLabelsForSubview:(UIView *)subview output:(NSMutableString *)output name:(NSString *)csvName,
  dumpImagesForSubview:(UIView *)subview tag:(int)tagtagArray:(NSMutableArray *)tagArray,
  dumpButtonsForSubview:(UIView *)subview tag:(int)tag tagArray:(NSMutableArray *)tagArray,
  dumpTableViewForSubview:(UIView *)subview tag:(int)tag tagArray:(NSMutableArray *)tagArray,
  dumpTableViewCellsForSubview:(UIView *)subview tag:(int)tag tagArray:(NSMutableArray *) tagArray

The dumpForSubview methods go through all UI element of the defined type (labels, images, buttons, tables and table cells) and define the required attributes for these specific types. Then they call the custom requirements methods to check these attributes.

- compareImagesForSubview:(UIView *)subview tag:(int)tag tagArray:(NSMutableArray *)tagArray

For every object from the CSV file with the available design asset, this method defines the associated UI element from the application on order to save and compare these two images. At first it compares the screenshot of the element and the asset using data comparison, and if the images are not similar, it compares them again using pixel-by-pixel comparison (Figure 12).

- savePNGForView:(UIView *)targetViewrect:(CGRect)rect filename:(NSString *)filename

This method defines the coordinates of the current UI element and saves its screenshot for the comparison. If the image is scaled and its size differs from the original asset size, this method resizes it to the size of the asset.

- imageWithImage:(UIImage *)image scaledToSize:(CGSize)newSize

This is a helping method for saving the current image, which renders the image representation with the declared width and height.

- getRGBA:(UIImage *)image ForX:(int)xx andY:(int)yy

To convert the color values to the readable and comparable RGB values, this method defines the red, green, blue and alpha values of the color, and calculates the RGB value in the appropriate format (for example 255/255/255, 63/78/83).

- convertPDFtoPNG

Some design assets are available only in the PDF format. This method converts the required assets from the PDF to PNG format for easier comparison, using PDFPageConverter and PDFPageRenderer classes.

- imageDataComparison

This method compares the images by comparing their byte information. This technique only works for the exact similar images that have not been scaled.

- imagePixelByPixelComparison

If the data comparison fails, the images will be compared using pixel-by-pixel comparison (Figure 13). To reduce time, it compares not every pixel but every 10th or 100th pixel, dependent on the size of the image, which is still enough for the pixel-by-pixel comparison. The algorithm calculates the number of all compared pixels with n = 1 for the small images and n = 10 for the large images: (width * height) / (100 * n * n).

```
1366 - (void)imageDataComparison{
1367
1368     NSLog(@"imageDataComparison");
1369     NSString *imageKorrekt = TEXT(@"ImageKorrekt");
1370
1371     if ([UIImagePNGRepresentation(self.image1) isEqualToData:UIImagePNGRepresentation(self.cvsAttributes.image2)] &&
1372         self.displayTrue == true){
1373         NSLog(@"%@", imageKorrekt);
1374         [self.output appendString: [NSString stringWithFormat:@"%@ \n",imageKorrekt]];
1375     }
1376
1377     else if (!([UIImagePNGRepresentation(self.image1) isEqualToData:UIImagePNGRepresentation(self.image2)]) && self.displayFalse
             == true){
1378         [self imagePixelByPixelComparison];
1379     }
1380 }
```

Figure 12: Implementation of image comparison function

```
1382 - (void)imagePixelByPixelComparison{
1383
1384     NSLog(@"imagePixelByPixelComparison");
1385     NSString *imageKorrekt = TEXT(@"ImageKorrekt");
1386     NSString *imageNichtKorrekt = TEXT(@"ImageNichtKorrekt");
1387
1388     // Pixel by Pixel comparison with 10% error tolerance
1389     int n = 1;
1390     CGImageRef imageRef1 = self.image1.CGImage;
1391     CGImageRef imageRef2 = self.image2.CGImage;
1392     int width = CGImageGetWidth(imageRef1);
1393     int height = CGImageGetHeight(imageRef2);
1394
1395     if ((width * height) > 80000){
1396         n = 10;
1397     }
1398     float numDifferencies = 0.0f;
1399     float totalCompares = (width * height) / (100 * n * n);
1400
1401     for (int yCoord = 0; yCoord < height; yCoord +=(10*n)){
1402         for (int xCoord = 0; xCoord < width; xCoord +=(10*n)){
1403
1404             NSArray *arrayA = [self getRGBA:self.image1 ForX:xCoord andY:yCoord];
1405             NSArray *arrayB = [self getRGBA:self.image2 ForX:xCoord andY:yCoord];
1406
1407             int valueA0 = [arrayA[0] intValue];
1408             int valueA1 = [arrayA[1] intValue];
1409             int valueA2 = [arrayA[2] intValue];
1410             int valueA3 = [arrayA[3] intValue];
1411
1412             int valueB0 = [arrayB[0] intValue];
1413             int valueB1 = [arrayB[1] intValue];
1414             int valueB2 = [arrayB[2] intValue];
1415             int valueB3 = [arrayB[3] intValue];
1416
1417             int img1RGB[] = {valueA0, valueA1, valueA2, valueA3};
1418             int img2RGB[] = {valueB0, valueB1, valueB2, valueB3};
1419
1420             if (abs(img1RGB[0] - img2RGB[0]) > 25 ||
1421                 abs(img1RGB[1] - img2RGB[1]) > 25 ||
1422                 abs(img1RGB[2] - img2RGB[2]) > 25){
1423                 numDifferencies++;
1424             }
1425         }
1426     }
1427
1428     if (numDifferencies / totalCompares <= 0.001f && self.displayTrue == true){
1429         NSLog(@"Image is right: %@", imageKorrekt);
1430         [self.output appendString: [NSString stringWithFormat:@"%@", imageKorrekt]];
1431     }
1432     if (numDifferencies / totalCompares <= 0.1f && numDifferencies / totalCompares > 0.001f){
1433         int percentage = (1 - (numDifferencies / totalCompares)) * 100;
1434         NSLog(@"Image is identical to %d percent", percentage);
1435         [self.output appendString: [NSString stringWithFormat:@"Image ist zu %d Prozent identisch.\n", percentage]];
1436     }
1437     else if (numDifferencies / totalCompares > 0.1f && self.displayFalse == true){
1438         NSLog(@"Image not right: %@", imageNichtKorrekt);
1439         int percentage = (1 - (numDifferencies / totalCompares)) * 100;
1440         NSLog(@"%@ Image is identical to %d percent \n", imageNichtKorrekt, percentage);
1441         [self.output appendString: [NSString stringWithFormat:@"%@ \nImage ist zu %d Prozent identisch.\n", imageNichtKorrekt,
                 percentage]];
1442     }
1443 }
```

Figure 13: Implementation of pixel by pixel comparison method

Then it detects the RGBA value for every 10*n pixel of both images, and saves it in the integer array for every image. Finally, it subtracts the red, green and blue values of the application image from the red, green and blue values of the asset. If the difference of at least one value is more than 25 (10 percent from 256 colors), then two pixels are different. For every false pixel, the number of differences is increased. When the relation of the number of differences to the total number of compared pixels is more than 10 percent, the images are not similar. So the method calculates the image difference with the error acceptance of 10 percent. It is important for the scaled images that may not be exactly the same, but still represent the same icon.

- customRequirements,
  compareCustomFamily,
  compareCustomFontSize,
  compareCustoFontColor,
  compareCustomBackgroundColor,
  compareCustomWidth,
  compareCustomHeight,
  compareCustomMarginLeft,
  compareCustomMarginRight,
  compareCustomMarginTop,
  compareCustomMarginBottom,
  compareCustomAlignment

These methods compare the required attributes of the UI elements with the customer design requirements from the CSV file.

- drawPageNbr:(int)pageNumber

The method calculates the page numbers of the resulted PDF document, in order to divide and render the output text to the correct page.

- updatePDF:(int)pageNumbersetTextRange:(CFRange *)pageRangesetFramesetter:(CTFramesetterRef *)framesetter

This method deals with the rendering of the output text to the PDF document with the testing results, using the declared settings.

- readCSVcorporate,
  readCSVWithParam:(int)tag

These two methods read the values of the corporate and customer design requirements from the comma separated CSV files and correlate them with the associated attributes (Figure 14).

```
2267 - (void)readCSVWithParam:(int)tag{
2268
2269     NSString *csvPath;
2270     NSLog(@"readCSVWithParam");
2271     NSLog(@"viewTag: %d", self.viewTag);
2272     NSLog(@"orientation: %@", self.orientation);
2273
2274     if (self.cvsAttributes.viewTag != 0){
2275         if([self.orientation isEqualToString:@"landscape"]){
2276             //csvPath = [NSString stringWithFormat:@"designvorgaben%d_landscape", self.viewTag];
2277             csvPath = [self.docs stringByAppendingString:[NSString stringWithFormat:@"designvorgaben%d_landscape.csv", self.
                     viewTag]];
2278         }
2279         if([self.orientation isEqualToString:@"portrait"]){
2280             //csvPath = [NSString stringWithFormat:@"designvorgaben%d_portrait", self.viewTag];
2281             csvPath = [self.docs stringByAppendingString:[NSString stringWithFormat:@"designvorgaben%d_portrait.csv", self.
                     viewTag]];
2282         }
2283     }

     <...>

2297     NSString *dataStr = [NSString stringWithContentsOfFile:csvPath
2298                                                    encoding:NSUTF8StringEncoding error:nil];
2299
2300     if (!dataStr) {
2301         NSLog(@"Error reading file CSV.");
2302     }
2303     NSArray *array = [dataStr componentsSeparatedByString:@"\n"];
2304
2305     for (int i = 0; i < ([array count] - 1); i++){
2306
2307         NSArray *itemArray = [[array objectAtIndex:i] componentsSeparatedByString:@";"];
2308         self.firstColumn = [itemArray objectAtIndex:0];
2309
2310         //if (![[itemArray objectAtIndex:0] isEqualToString:@"tag"]){
2311         self.csvTag = [self.firstColumn floatValue];
2312         if (self.csvTag == tag){
2313
2314             // Design/Asset Name
2315             self.csvName = [itemArray objectAtIndex:1];
2316             NSLog(@"CSV NAME: %@", self.csvName);
2317
2318             // 2 - Maske/Screen
2319             // 3 - Asset-Typ
2320
2321             self.csvAssettyp = [itemArray objectAtIndex:3];
2322
2323             // 4 - Filetyp
2324
2325             // FileName
2326             self.csvAsset = [itemArray objectAtIndex:5];
2327
2328             // Width 1
2329             if([itemArray objectAtIndex:6] != nil){
2330                 NSNumber *cBreite = [itemArray objectAtIndex:6];
2331                 self.csvBreite = [cBreite floatValue];
2332             }
2333             else if([itemArray objectAtIndex:6] == nil){
2334                 self.csvBreite = 0;
2335             }
2336
2337             // Height 1
2338             if ([itemArray objectAtIndex:7] != nil){
2339                 NSNumber *cHoehe = [itemArray objectAtIndex:7];
2340                 self.csvHoehe = [cHoehe floatValue];
2341             }
2342             else if ([itemArray objectAtIndex:7] == nil){
2343                 self.csvHoehe = 0;
2344             }
```

Figure 14: Implementation of reading the design requirements function

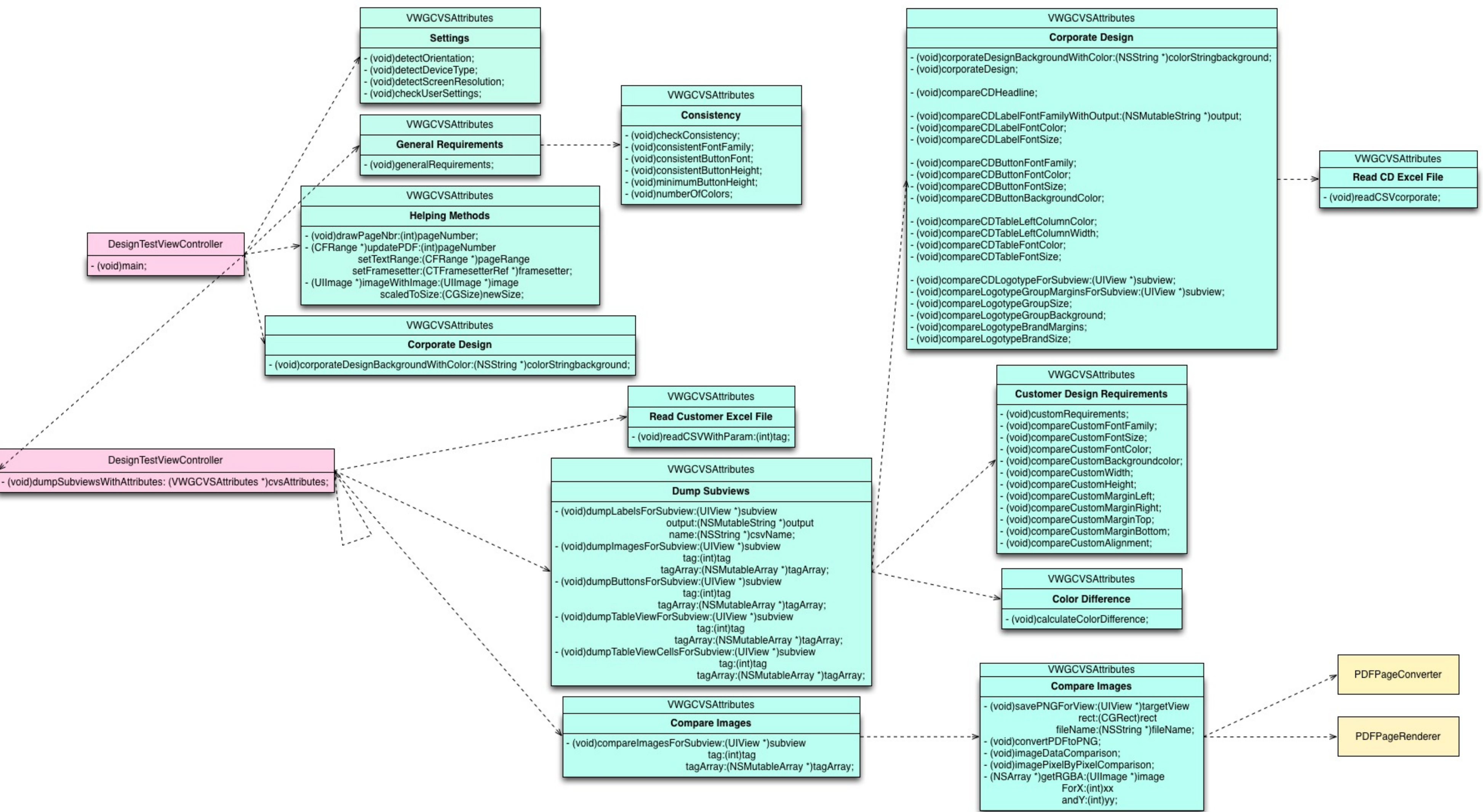

Figure 15: System method architecture

5.4. Examples

5.4.1. iAgree

iAgree is a mobile application for iPad or iPhone that could be used instead of lot of papers the employees has to work with every day (Figures 16 and 17). The application permits the making of processes approval tasks in the train, airplane, or elsewhere without using paper documents. It makes it possible to load the documents from the repository, read them on the mobile device, and easily approve or decline them with a few clicks. If the device is in an offline mode, the application saves all changes and loads them when it is online again. iAgree does not replace the original workflow of the approval process but enables the handling of approval tasks centrally. In addition to approval and decline processes, there is an opportunity to choose a selection from more alternatives. The processes added to the system can also be deleted from iAgree, for example when the user handled the tasks in the original system and not in iAgree.

In order to test the design requirements of iAgree, the DesignTesting framework was linked to the iAgreeMock target of the application project. The design of the application was developed by Artlab Studios Berlin, and in addition to screenshots, the values were also represented in the Excel table. So only few modifications of the design requirement document were needed, including the adding of element ids. The Excel document with the customer design requirements is shown in the figure 18.

iAgree belongs to Volkswagen Group applications, so for the testing of its corporate design the user preferences must be set to company corporate design in the device settings section. The CVS files of the design requirement documents and assets are loaded in iTunes. When the user starts the application and shakes the device, the code is executed and the result PDF document is created in iTunes. Executing of the code takes 12 seconds. Most of this time is spent on the image pixel-by-pixel comparison. The result document (Figure 19) includes the information about VW logotype and about all components of the main screen. When the user opens the vertical slider with settings and shakes iPad again, the system creates a new PDF document with the information about buttons and labels in the slider. So it is possible to test every view of the application. According to the result of the test, the logotype of the application uses the right image, but the width of the image is 121 pixels instead of 120 pixels. Also the width of the left column is 319 pixels, instead of 320 pixels. According to the Design Testing Framework, the corporate design of the application fails, but since all the values are represented in the output document, the testers can decide manually whether the difference of 1 pixel is essential for the corporate design. Concerning the typeface attributes, most text elements in iAgree use Verdana font family and blue color (76/83/86), which are allowed for the company corporate design, however some colors are different. Nevertheless, iAgree uses different font sizes for various text elements. The explanation for this could be that different types of labels (title, subtitle etc.) can have different attributes, so more differentiation of the element types is needed in the design requirements and in the testing tool. All customer requirements are defined and compared correctly.

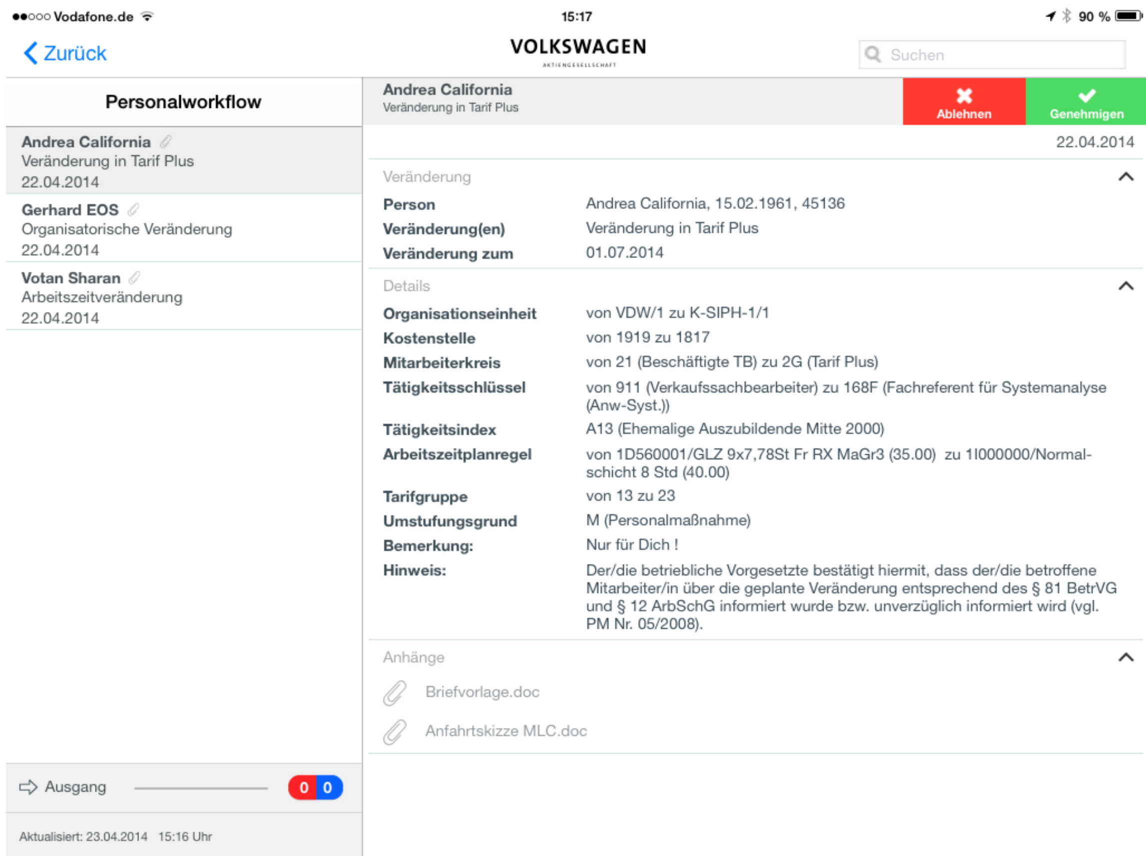

Figure 16: Screenshot of iAgree main view with accept and decline options

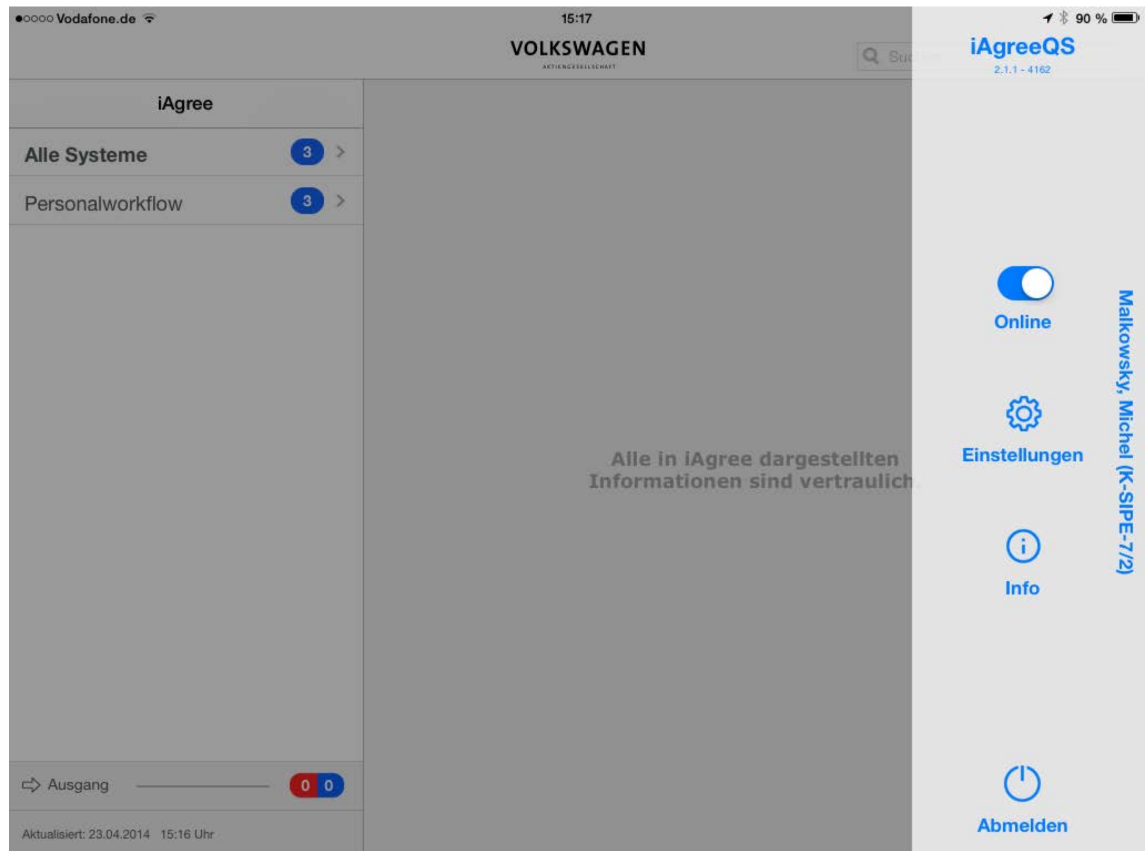

Figure 17: Screenshot of iAgree view with opened vertical slider

	Tag	Design-Asset	Maske/Screen	Asset-Typ	Filetyp	FileName	Breite1	Hoehe1	Breite2	Hoehe2
10	9	Medium Label - Label	Haupt-Screen	Text			179	22		
11	10	Button - Confirm Selection	Haupt-Screen	Button			104	32	208	64
12	11	Button - Approve Button	Haupt-Screen	Button			104	32	208	64
13	12	Button - Cancel	Haupt-Screen	Button			104	32	208	64
14	13	Large Label - Title	Haupt-Screen	Text			160	25		
15	14	Button - Back	Haupt-Screen	Button		BackButton.pdf	74	37	148	74
16	15	Header - Volkswagen-Akrien	Allgemein	Icon	Vektor-PDF	VW-Logo.pdf	121	25	242	49
17	16	ImageView Search	Haupt-Screen	Icon	Vektor-PDF	Searchfield_full.pdf	240	26	480	52
18	17	Medium Label - Label	Haupt-Screen	Text			433	25		
19	18	Medium Label - Label	Haupt-Screen	Text			644	21		
20	19	Small Bold Label - Label	Haupt-Screen	Text			173	25		
21	20	Label - Outbox	Haupt-Screen	Text			63	20		
22	21	ImageView	Haupt-Screen	Icon	Vektor-PDF		17	14		
23	22	Button	Haupt-Screen	Button			104	32	208	64
24	23	TableView	Haupt-Screen				320	571		
25	24	Label linke Spalte	Haupt-Screen	Text			295	28		

Figure 18: Design requirements for iAgree presented in the Excel table

____

Tag: 50
Label Name: Label - All information presented in iAgree are confidential.

CUSTOM REQUIREMENTS:

---Font Family---
Excel Font Family: Verdana
App Font Family: Verdana
Schriftart ist RICHTIG!

---Font Size---
Excel Font Size: 17
App Font Size: 17
Schriftgröße ist RICHTIG!

---Font Color---
Excel Font Color: 166/166/166
App Font Color: 166/166/166
Schriftfarbe ist RICHTIG!

CORPORATE DESIGN:

CD Schriftart ist RICHTIG: Verdana
CD Schriftfarbe ist FALSCH: 166/166/166. Schriftfarbe muss 76/83/86 sein.
CD Schriftgröße ist FALSCH: 17 px. Schriftgröße muss 15 px sein.
____

Figure 19: Results of the design testing of iAgree

5.4.2. Konzernkalender

Konzernkalender is the mobile solution for representing the most important events of the year in a personal calendar. It also gives the opportunity to subscribe peripheral created calendars with different types of events and represent them in your own calendar view. The application allows the editing of the calendar through changing the color, marking and hiding or displaying different calendars. The events can be shown in the year, quarter or month view (Figures 20 and 21). The calendars can be controlled in a web-based editor tool. Only authorized people can create and edit events in order not to overfill the calendar with redundant information. The data is transferred through encrypted and safe connections. The authorization is given via Volkswagen identification and by entering the password. It was superficially considered that the interface of the Konzerkalender application should be designed for ease of use and should not distract users from the essential information.

All layout information of the Konzernkalender application is designed in the main.storyboard file in Interface Builder, as well as in some XIB files, which can be also seen and edited in the Interface Builder. All customer design requirements were also entered in the Excel template (Figure 22).

The design testing of Konzernkalender was done in the same way through the linking of DesignTesting Framework, loading resources in iTunes and building the KK_NOPKI target. The user can choose one of the calendar views (year view, quarter view or month view) and shake the device. The system creates an output document, with the information about all elements represented in this particular view (Figure 23). Execution of the code takes 17 seconds. Konzernkalender also belongs to Volkswagen company applications. While most customer requirements are followed in the application, the corporate design does not correspond to the guidelines. Looking at the outputted result, it can be concluded that almost all found elements and attributes were defined and compared correctly, but there are some inaccuracies. The logotype has a width of 300 pixels, although its width is looking like it required 120 pixels. It is because of this that the original image already contains the white spacing from the sides. So the source code analyses can only define the size of an image asset and not the size of the real image inside this asset. This can be better done with the image recognition technique. In contradiction to iAgree, the UI elements presented in pop-up windows, for example with the information about a certain event, do not appear in the result PDF. So it is possible to make a conclusion that the way in which the application is implemented has an influence on the testing result. It means that the design testing tool must be aware of different application structures and different ways of implementation.

●●●○○ Vodafone.de 3G 11:20 81 %

Kalender 2013 VOLKSWAGEN Monat Quartal Jahr

Figure: 20: Screenshot of Konzernkalender year view

●●●○○ Vodafone.de 3G 11:20 81 %

Oktober 2013 VOLKSWAGEN Monat Quartal Jahr

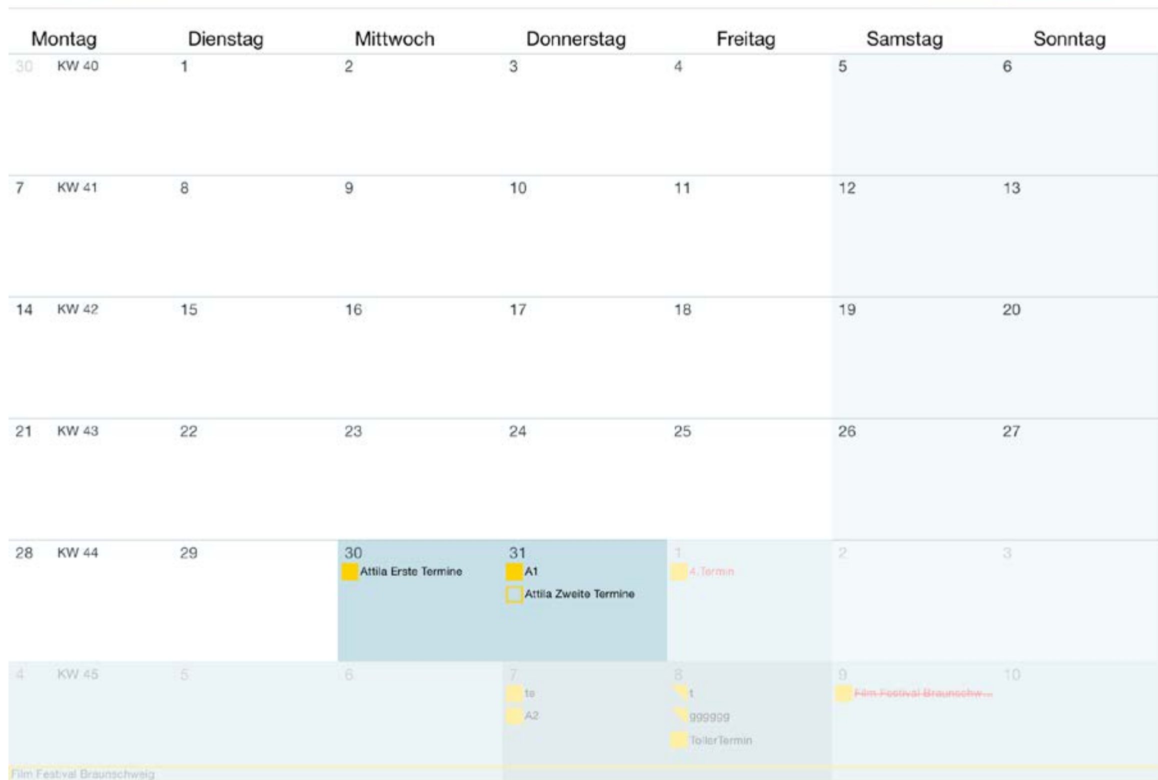

Figure 21: Screenshot of Konzernkalender month view

Farbcode fuer Flaechen (RGB)	Effekte	Text - Groesse	Text - Farbcode (RGB)	Text - Font	Text - Ausrichtung
198/223/231					
171/199/209					
146/171/187					
		14	255/0/0	Helvetica Neue	Left
		14	255/0/0	Helvetica Neue	Left
255/128/0					
		22	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left
		14	0/0/0	Helvetica Neue	Left

Figure 22: Design requirements for iAgree represented in the Excel table

GENERAL REQUIREMENTS
____

Text Schriftart ist NICHT überall gleich!
Button Schriftart ist überall gleich!
Button Schriftgröße ist NICHT überall gleich!
Button Höhe ist NICHT überall gleich!

Konsistenz wird NICHT eingehalten.

Anzahl der Farben ist RICHTIG. Die App enthält weniger als 7 verschiedenen Farben.

Die Button Höhe ist FALSCH. Sie muss mindestens 35 px sein.

CD Hintergrundfarbe ist RICHTIG: 255/255/255

____

Tag: 49
Image ist RICHTIG.
Image Name: VW Logo

CUSTOM REQUIREMENTS:

---Width---
Excel Width: 300
App Width: 300
Breite ist RICHTIG!

---Height---
Excel Height: 45
App Height: 45
Höhe ist RICHTIG!

CORPORATE DESIGN:

Abstand der Wortmarke von links ist RICHTIG.

Figure 23: Results of the design testing of Konzernkalender

5.5. Limitations

The results of the testing of two applications show that my DesignTesting Framework can efficiently determine the visible UI elements and their attributes and can reduce the time effort of the testing process. However, the developed prototype contains some limitations that can be improved in the next version of the tool. Depending on how the application under test is implemented, some UI elements visible on the current screen cannot be recognized with the tool. On the other hand, the system sometimes finds components that are implemented for this view, but not visible on the screen at the moment, because they have to be activated. The possible solutions of this problem include the awareness of different ways of implementation, setting the rules for the developers to realize the applications in the same way, or combining the source code analysis method with the image recognition technique.

In addition, some elements can be designed using different view types. For example, text components can be made as labels, text views or even as images. The same method can also be applied for buttons –they can have a background and a title, but can also be designed as an image. To settle this issue, the tool already uses the image comparison of the element and the asset, but in some cases the asset may not be available.

Another limitation is that DesignTesting Framework at the moment considers only some types of the elements – labels, buttons, images and tables. It can distinguish between different types of text components (normal text, title, text in a table cell) in the code and through defining them in a special column of the design requirements table. But to achieve more accurate testing results there is a need of more differentiation between variousUI element types. The tool lacks the information about such UI components as tab bar, navigation bar, sliders, radio buttons, footer and others.

The design-testing prototype is working dynamically and finds the current information about the application at the running time. Nevertheless, there are can be some difficulties with the dynamically changing elements, because they are statically described in the design requirements document. Excel table is able to calculate the values in cells dependent on other cells, for example the logotype spacing from left and right dependent on the width of the logotype. But there are some variables that depend on the values not described in the design requirements document, such as current screen resolution or scaling of the view. Additionally, some constraints are described in the Interface Builder in relation to certain objects. So the button can have minimal spacing of 20 pixels from the navigation bar, from the header or from the closest button. Now the design-testing prototype can only define the spacing from the window sides, and does not have an opportunity to describe and test the considerations in relation to other components. The dynamic attributes and dependencies on other elements can be solved through reorganizing the design requirements document, adding more complex functionality to the source code, or through involving the image recognition of the screen elements.

6. User study of DesignTesting Framework

6.1. Analysis of the target group

Before I could start to design the user study, I needed to know what kind of persons I want to test. Since the design requirements tested in the experiment are developed in the customer management department of Volkswagen Group AppFactory, and my Design-Testing Framework should be used in the future in the quality management department, the persons of the target group should have experience working with the potential customers or testing the applications. I tried to find both female and male participants to make the test more consistent. Furthermore, the test persons need to be familiar with using computers and mobile devices, since they are used in the experiment.

The execution of the automated design testing tool always takes the same time and gets the same results, so the number of participants is not so essential. More important is the testing of different cases and different attributes to simulate as many possible scenarios of use as possible. I chose for the experiment 3 participants, one male and two females between the ages of 18-34 years. All of them work in Volkswagen Group AppFactory, one in the customer management and two in the quality management departments. In addition, all of them have had some experience with testing mobile applications and have already worked before with HP ALM, a software used in my experiment.

Test person #1 is a female in the age range of 18-24 years, who graduated from high school and is now studying media computer science. She is doing an internship in the quality management department of Volkswagen Group AppFactory, and was engaged in the last six months with testing tasks. Test person #2 is also a female in the age range of 25-34 years with a bachelor degree in computer science. She is working as a software tester in the quality management department. Test person #3 is a male in the age range of 25-34 years old with a bachelor degree in information management. He is doing an internship in the customer management department. He also has some experience with software testing, but less than the other two participants.

6.2. Procedural method

The independent variables of the experiment were test environment, the applications and the test cases. They have not changed during the whole test. The dependent variable was the method of testing – manually and automated.

The quantitative data collected in the study were duration of each test case and the number of wrong and correct answers. The time was measured with the help of a stopwatch. The answers of the test persons were recorded in the ALM, and were then manually compared with the real results in the application according to the table. The qualitative data collected during the experiment was the satisfaction of the participants with the tool. It was measured with a help of the usability questionnaire, comments of the test persons, and observational notes.

6.3. Experiment design

The goal of the experiment was to prove how the use of DesignTesting framework affects the productivity and duration of the test process. For this purpose, the test persons had to test two applications both manually and with the help of the automated tool. The user study includes four test cases – testing of two views of each of two applications. Each test case consists of several steps. First, these test cases were done manually, and then the same test cases were done automatically.

The experiment took place in the quality management laboratory of Volkswagen Group AppFactory. A hardware used in the experiment included a Windows computer, with installed HP ALM and all standard software, MacBook Pro, with installed XCode and iTunes, and iPad 3 with a cable connected to MacBook. ALM is used for the easier creation of test cases and recording the test results. Both application projects were located in XCode and were prepared for the run. iTunes was used for loading the required resources and saving the results. The test environment for all participants was the same. In the experiment, two applications described before were used – iAgree and Konzernkalender. The participants were tested after each other with the same conditions.

When the test subject and the use of DesignTesting Framework were explained to the test persons, they were asked to complete four test cases described in the ALM. The test cases with all steps and expected results are represented in the table 3. Each test case had to be done both manually and with the use of the automated tool. For the manual testing, the participants could use all possible methods. They had to define the attributes of the stated elements as they would do in their usual work if they needed to test the design of an application. They could use the screenshots of the application and different software, such as Microsoft Word, Paint or Internet Explorer to define the font family, sizes or colors. Thus, the participants were free as to how they determined the required attributes. All results were marked in ALM. Then the test persons had to complete the same four test cases with the use of my automated tool. First, they had to run the application with the linked Design-Testing Framework on iPad, to shake the device and to wait till the code was executed. Then they could open the generated PDF document, which was located in iTunes, and read the results. According to these results, they could mark the described test cases as right or failed. The whole experiment took approximately 60-70 minutes per person. During the test, I took notes about my observations and the person's comments.

At the end the participants were asked to fill out the online questionnaire. It was based on the combination of questions from four questionnaires - Perceived Usefulness and Ease of Use Questionnaire, Questionnaire for User Interface Satisfaction, Computer System Usability Questionnaire and QUESI. The questionnaire consisted of 5 parts, including demographical questions about the participants and their background, questions about usefulness, ease of use, satisfaction and opened questions about negative and positive features of the tool. To answer the questions in the usability and satisfaction parts, the test persons had to choose a number on the likert scale from 1, which means "strongly disagree", to 5, which means "strongly agree". The numbers they choose should match their level of agreement with the statements about the tested tool.

Step	Task	Expected result	Result according the application
Test case 1: Test the main view of iAgree			
Step 1	Check the logotype image	Image is true	True
Step 2	Check the width of the logotype image	120 px	False
Step 3	Check the spacing of logotype image from left and right	At least half of the logotype width	True
Step 4	Check background color of the logotype image	White (255/255/255)	True
Step 5	Check search field image	Image is right	True
Step 6	Check the font family of the text "All information presented in iAgree are confidential"	Verdana	True
Step 7	Check the font size of the text "All information presented in iAgree are confidential"	17 pt	True
Step 8	Check the font colors of the text "All information presented in iAgree are confidential"	Grey (166/166/166)	True
Step 9	Check the font family of the text in the left column	Verdana	True
Step 10	Check the font size of the text in the left column	17 pt	True
Step 11	Check the font color of the text in the left column	VWAG Grey (76/83/86)	True
Step 12	Check the width of the left column	320 px	False
Step 13	Check the background color of the left column	VWAG Petrol light 50% (231/243/243)	True
Step 14	Check the font family of the Outbox label	Verdana	True
Step 15	Check the font size of the Outbox label	14 pt	True
Step 16	Check the font color of the Outbox label	VWAG Grey (76/83/86)	True
Step 17	Check the consistence of the main screen view	Font family of all text elements is the same	False
Step 18	Check the number of different colors in the main screen view	Less than 7colors	False
Step 19	Check the corporate design of the main screen view	Font family: Thesis TheSans, Verdana or FF Cellini, Font size: 15 pt, Font color: VWAG Grey (76/83/86)	False
Test Case 2: Test the vertical slider of iAgree			
Step 1	Check the font family of the application name label	Helvetica Neue	True
Step 2	Check the font size of the application name label	21 pt	True
Step 3	Check the font color of the application name label	Blue (6/72/102)	False
Step 4	Check the font family of the application version label	Helvetica Neue	False

Step 5	Check the font size of the application version label	9 pt	True
Step 6	Check the font color of the application version label	Blue (6/72/102)	False
Step 7	Check the font family of the settings label	Helvetica Neue	True
Step 8	Check the font size of the settings label	17 pt	True
Step 9	Check the font color of the settings label	Blue (6/72/102)	False
Step 10	Check the settings button image	Image is right	True
Step 11	Check the size of the settings button	44x44 px	True
Step 12	Check the font family of the info label	Helvetica Neue	True
Step 13	Check the font size of the info label	17 pt	True
Step 14	Check the font color of the info label	Blue (6/72/102)	False
Step 15	Check the info button image	Image is right	True
Step 16	Check the size of the info button	44x44 px	True
Step 17	Check the font family of the logout label	Helvetica Neue	True
Step 18	Check the font size of the logout label	17 pt	True
Step 19	Check the font color of the logout label	Blue (6/72/102)	False
Step 20	Check the logout button image	Image is right	True
Step 21	Check the size of the logout button	44x44 px	True
Step 22	Check the font family of the user name label	Helvetica Neue	True
Step 23	Check the font size of the user name label	17 pt	True
Step 24	Check the font color of the user name label	Blue (6/72/102)	False
Test case 3: Test the year view of Konzernkalender			
Step 1	Check the logotype image	Image is right	True
Step 2	Check the width of the logotype image	120 px	Unknown
Step 3	Check the spacing of the logotype image from left and right	At least the half of the logotype width	True
Step 4	Check the background color of the logotype	White (255/255/255)	True
Step 5	Check the font family of the calendar name	Helvetica Neue Interface	True
Step 6	Check the font size of the calendar name	24 pt	False
Step 7	Check the font color of the calendar name	Black (0/0/0)	False
Step 8	Check the font family of the month name	Helvetica Neue Interface	True
Step 9	Check the font size of the month name	20 pt	True
Step 10	Check the font color of the month name	Black (0/0/0)	True
Step 11	Check the consistency of the year view	font family of all text elements is the same, font family of all button titles is the same, font size of all button titles is the same, height of all buttons is the same	False
Step 12	Check the background color of the year view	White (255/255/255)	True
Step 13	Check number of different colors in the year view	Less than 7 colors	True
Test case 4: Test the month view of Konzernkalender			
Step 1	Check the font family of the week day label	Helvetica Neue	False
Step 2	Check the font size of the week day label	17 pt	True

Step 3	Check the font color of the week day label	Black (0/0/0)	True
Step 4	Check the font family of the day label	Helvetica Neue	True
Step 5	Check the font size of the day label	14 pt	True
Step 6	Check the font color of the day label	Black (0/0/0)	False
Step 7	Check the font family of the calendar week label	Helvetica Neue	False
Step 8	Check the font size of the calendar week label	11 pt	False
Step 9	Check the font color of the calendar week label	Black (0/0/0)	False
Step 10	Check the font family of the event name label	Helvetica Neue	False
Step 11	Check the font size of the event name label	18 pt	False
Step 12	Check the font color of the event name label	Black (0/0/0)	False

Table 3: Test cases for the research experiment

6.4. Data representation

During this study, the quantitative (time, results of test cases, usability questionnaire) and qualitative (opened questions, observational notes) types of data were collected.

The time taken for each experiment manually and with the use of the automated tool is represented in the table 4. For the first two test cases (application iAgree), there is a tendency that the automated testing took less time than the manual testing. The third and fourth test cases (application Konzernkalender) had fewer steps and took in general less time. In these test cases the manual testing took less time than with the use of Design-Testing Framework, though the difference is very small. Test person #1 completed all manual tests in 34 minutes and all automated tests in 30 minutes. Test person #2 spent in total 33 minutes for the manual test and for the automated test only 25 minutes. Finally, test person #3 finished the manual testing in 30 minutes and automated testing in 32 minutes. In total, the time used for all manual tests by all three participants was 97 minutes and for all automated tests 87 minutes, which is 89.7 percent of the time spent for manual testing. That means that the use of Design-Testing Framework can reduce the time of the testing process to 10.3percent.

Another type of quantitative data collected during the experiment was the number of correct and wrong answers. Every step of the test case could be either right or wrong, dependent on whether the attribute value expected in the test correlated to the value in the application. It is presented in the table 5. If the answer given by test person (passed or failed) matched the answer declared in the table, it was seen as a correct answer; if it does not match the answer given in the table, it was seen as an error. The table below presents a number of errors for each test case. In manual testing, no cases were done without errors. Half of test cases done with the use of the automated tool were done without errors, the others had only a small number of errors. The total number of errors done during the manual testing by all participants is 73, while the number of errors done during the automated testing is only 12.

Test case	Type of test	Test person #1	Test person #2	Test person #3
Test case 1	Manually	13 minutes	13 minutes	13 minutes
	Automated	9 minutes	9 minutes	14 minutes
Test case 2	Manually	12 minutes	13 minutes	9 minutes
	Automated	8 minutes	6 minutes	7 minutes
Test case 3	Manually	5 minutes	4 minutes	5 minutes
	Automated	6 minutes	5 minutes	4 minutes
Test case 4	Manually	4 minutes	3 minutes	3 minutes
	Automated	7 minutes	5 minutes	7 minutes

Table 4: Duration of every test case for both manual and automated design testing

Test case	Type of test	Test person #1	Test person #2	Test person #3
Test case 1	Manually	7 errors	9 errors	4 errors
	Automated	0 errors	1 error	0 errors
Test case 2	Manually	16 errors	9 errors	2 errors
	Automated	0 errors	0 errors	2 errors
Test case 3	Manually	3 errors	2 errors	3 errors
	Automated	1 error	2 errors	2 errors
Test case 4	Manually	6 errors	5 errors	7 errors
	Automated	4 errors	0 errors	0 errors

Table 5: Number of errors for every test case for both manual and automated design testing

The usability questionnaire was used to measure the usability of the tool and the overall satisfaction with the system. The questions of the questionnaire were combined in three categories with similar characteristics to narrow the data – usefulness, ease of use and satisfaction. The answers were given on a 5-point likert scale, where 1 corresponded to "strongly disagree", 2 to "disagree", 3 to "neither agree nor disagree", 4 to "agree" and 5 to "strongly agree". Although there were only three test persons, and this is not enough to analyze results statistically, I describe the general results of the main points shortly to give a first overview. The questionnaire with the average answers for all questions is presented in table 6. The "usefulness" part of the questionnaire demonstrated how useful the automated tool could be in increasing the productivity of the design testing process. The participants stated that my DesignTesting Framework could be useful for the testing of design requirements by an average value of 4.13, which is the highest ranking among other categories. It means they agree that using the automated tool can improve the testing of design requirements. The "ease of use" part showed how understandable, intuitive and learnable the system was. The average value for this category is 3.95. The statement with the highest average answer of 4.67 was "The system was easy for me to use". The lowest answer of 2.67 was given for the statement "I could use the system without any instructions". So it would not be automatically clear how to use the tool without documentation, but since before the test the guidelines were given, it was easy enough to understand how it works. The "satisfaction" part of the questionnaire demonstrated the overall impression of the tool. The participants stated that they are satisfied with the system by an average value of 3.84, which is slightly under the "agree" option in the ranking.

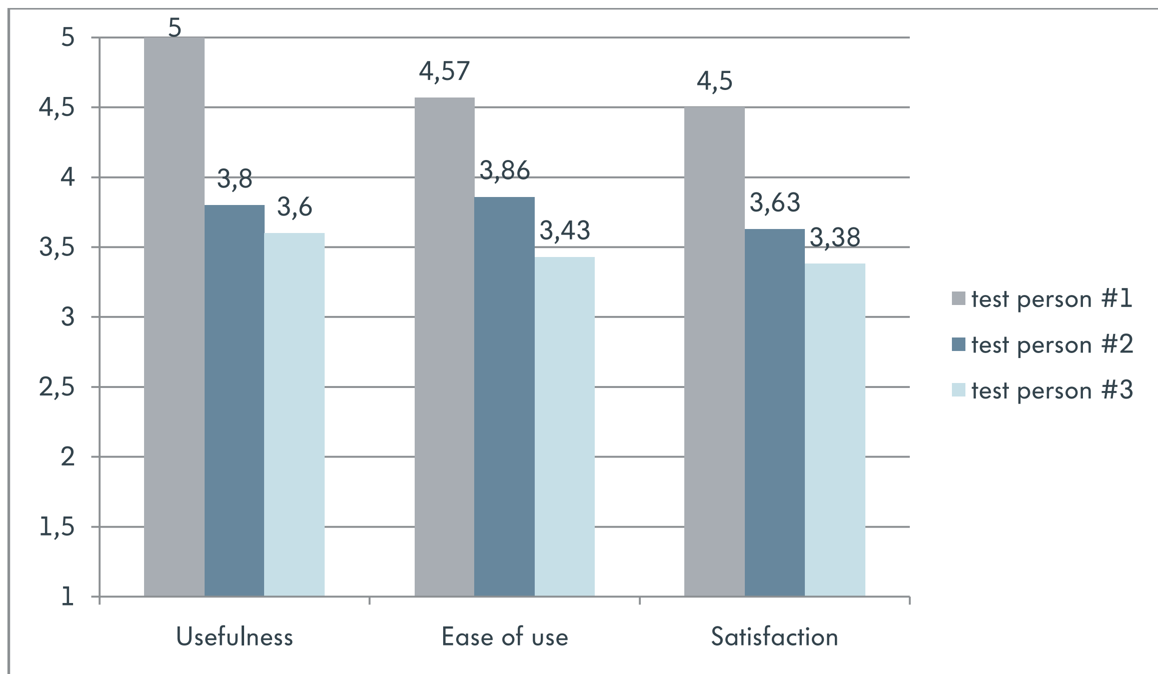

Figure 24: Results of the usability questionnaire

The distribution of the answers of each test person to the questionnaire is presented in the diagram in figure 24.

These answers show the main tendency in the participant's feelings about the system after the test, but for more accurate results they should be combined with the qualitative data from open questions and observational notes. Thus, none of the test persons has mentioned any negative aspects about the system. One of the test persons wrote in the "positive aspects" section that the tool was "easy to use and easy to understand". The participants were also asked to list additional functionalities they would expect from the system. Two of them answered that "PDF outcome of the system could be more sorted, for example in a table view" and that the system could use "more obvious keywords in the PDF document". So both improvement proposals deal with the representation of the test results in the PDF document.

The observational notes taken during the experiment demonstrate what means the participants used to find out the attributes of the UI elements. Test person #1 used mainly Microsoft Word to define the typefaces, Paint to define the sizes and spacing and Internet Explorer to define the colors of the elements. Test person #2 used Internet Explorer to define all types of attributes, and sometimes Microsoft Word to find out the typefaces. Test person #3 checked almost all attributes in Paint: he drew the declared elements with the expected attributes over the screenshot of the application. In the manual testing, the participants often had difficulties with defining the attributes, so the values of the attributes were often guessed. They gave comments like "I can imagine that it is true, but I cannot be totally sure".

Usability Questionnaire					
Question number	**Statement**	**Test person #1**	**Test person #2**	**Test person #3**	**Average result**
Usefulness					
1	Using the system for testing tasks would enable me to finish my job more quickly	5	4	4	4,33
2	Using the system would improve my job performance	5	4	4	4,33
3	Using the system would increase my productivity	5	3	3	3,67
4	Using the system would make it easier to get done the tasks I want to accomplish	5	4	3	4,0
5	I would find the system useful for the testing tasks	5	4	4	4,33
Ease of use					
6	The system was easy for me to use	5	5	4	4,67
7	Learning to operate the system would be easy for me	5	4	4	4,33
8	I would find it easy to get the system to do what I want to do	3	4	3	3,33
9	I could use the system without instructions	4	2	2	2,67
10	The system requires fewest steps possible to accomplish the tasks I want to do with it	5	4	4	4,33
11	I didn't notice any inconsistencies using the system	5	4	4	4,33
12	I automatically did the right steps to achieve my goals	5	4	3	4,0
Satisfaction					
13	Overall, I am satisfied with how I could use the system	5	4	4	4,33
14	I could effectively complete my tasks using this system	5	4	3	4,0
15	No problems occurred when I used the system	5	4	4	4,33
16	I feel comfortable using this system	4	4	4	4,0
17	I would recommend to use this system for testing tasks	4	4	4	4,0
18	The system is pleasant to use	4	4	3	3,67
19	The system meets my needs	4	3	2	3,0
20	The system has all functions and capabilities I expect it to have	5	2	3	3,33

Table 6: Usability questionnaire with average results

In addition, the observation and participant's comments demonstrate what difficulties they had testing the design requirements with the automated tool. Test person #1 had problems at least in finding results in the PDF document, for test person #3 it was most difficult. All three participants used a search function to find the elements in the PDF document: in most cases they could find them very quickly, but sometimes the names of the elements were not obvious enough and they were not sure what element was meant. Also the difference between customer and corporate design requirements was not always clear and they did not know from which section they should take the result. Test person #2 said that

the results in the PDF document should be more visual. They could be represented in the table like the design requirements itself, but in the cell, according to the certain element, and attribute should be written whether it true or false. The output could also have images and the order of the elements could be more logical. All three test persons tested the second applications more quickly and not so accurately. In general, they could easily use the automated tool, with the instructions given before, and evaluate the results, except there were some difficulties with the naming of the elements and representation of the results.

6.5. Data analysis

The evaluation is based on the data of 3 participants. The number is surely too small to make a meaningful quantitative analysis: that's why it makes sense to take into account the answers of every participant for the qualitative analysis. All three participants use computers and mobile devices every day and have already had experience with the testing of applications; however, only two of them are engaged with the application testing in their usual work.

Test results of test person #1 are shown in table 7. Test person #1 is a student, currently working on the testing of mobile applications in the quality management department of AppFactory. She looked very competent with the testing process and could complete all steps without help. In the manual testing of design requirements she had some difficulties with defining the attributes of the elements. It took a long time to compare the colors, sizes and typefaces on iPad with those found in Internet or in Word. She often said that she could imagine that it was true, but could not define the value exactly. As a result, she sometimes just assumed that the attributes were right or not. For testing the second application, test person #1 needed much less time. Sometimes she did not compare the attributes with the values found with different helping means and checked them directly as passed or failed. The reason for this could be that she was already familiar with how the repeating attributes look like and did not need to compare them again. Another reason could be that she was tired after the testing of the first application and wanted to finish the test fast. Both cases can lead to the inattentive testing of every single element, so that small distinctions in the values can be overlooked and errors may occur. During the testing of the applications test person #1 made 23 errors in 43 tasks in iAgree and 9 errors in 25 tasks in Konzerkalender, while checking them manually. The amount of errors in the first application is thereby more than 50 percent. Using the DesignTesting Framework she made no errors in iAgree and 5 errors in Konzernkalender. So testing the applications manually caused many more errors than using the automated tool. Test person #1 spent 8 minutes more for testing the first application manually than with the use of tool. For the second application she needed 4 minutes less for the manual testing. In general, it was much easier to find out the results of the test doing it with the help of the automated tool. The only problem was the representation of results in the PDF document. The names of some elements were not self-descriptive enough, but these names come from the Excel table created by designers. So the rules for better naming of objects in the design requirements and in the application's code should be created in the future. In addition, the difference between customer and corporate design requirements was not clear, and different values in both categories for the same element were irritating. This is because in practice customer requirements do not always correlate with corporate design

Test person	Introduction questions	Time	Number or errors	Observation, comments	Usability questions
#1	Female, 18-24 years old, student in media computer science, internship in quality management, uses computer and mobile devices every day, has experience in testing	Manual testing: iAgree: 25 min KK: 9 min Automated testing: iAgree: 17 min KK: 13 min	Manual testing: iAgree: 23 errors KK: 9 errors Automated testing: iAgree: 0 errors KK: 5 errors	Manual testing: Used Word, Paint and Internet to define attributes, "I can imagine that it is true, but it is difficult to define the value precisely enough". Automated testing: Difference between customer and corporate design is not clear, had no problems with finding out the attributes, but some problems with finding elements in PDF output, "more obvious keywords in the PDF document"	Usefulness: 5.0 Ease of use: 4.57 Satisfaction: 4.5

Table 7: Experiment results of test person #1

guidelines, and the tester should decide before the test which requirements are more important for this case. In general, test person #1 gave the highest rating to the questions in the usability questionnaire and found the system useful (5.0) and easy to use (5.0).

Test results of test person #2 are represented in table 8. Test person #2 works in the quality management department with software testing and has much experience with the testing of mobile applications. Nevertheless, she is engaged with functionality and usability testing and not with design testing. Like test person #1, she had difficulties with defining the attributes of the elements in the manual testing. It was especially hard for her to define the sizes of the elements. Test person #2 made 18 errors in iAgree and 7 errors in Konzernkalender while doing the manual testing of applications. Using Design-Testing Framework she made only 1 error in iAgree and 2 errors in Konzernkalender. So, here too, it is evident that automated testing has caused many fewer errors. In the case of the duration of the test process with regard to test person #1, the same tendency can be seen. The testing of the first application manually took 11 minutes more than doing it automatically, while the testing of the second application was done 3 minutes slower using the tool. Test person #2 had the same problems while working with the output document. It was difficult to find some elements, and she had to scroll the document a lot. The difference between customer and corporate requirements was clear only after explaining it. Test person #2 suggested representing the results in another way. The elements could be sorted by the name or the type of the object. To enable the better overview of all elements, they could be presented in the table like the values in the design requirements document. The names of the elements could be located on the left and the attributes on the top. In the intersecting cell, it could be written whether this attribute of this element is correct or not. Also the images of the elements could be shown in order to find them faster. Independent

Test person	Introduction questions	Time	Number or errors	Observation, comments	Usability questions
#2	Female, 25-34 years old, working in software testing, uses computer and mobile devices every day, has experience in testing	Manual testing: iAgree: 26 min KK: 7 min Automated testing: iAgree: 15 min KK: 10 min	Manual testing: iAgree: 18 errors KK: 7 errors Automated testing: iAgree: 1 error KK: 2 errors	Manual testing: Used Word and Internet to define attributes, had difficulties with defining the attributes Automated testing: problems with finding elements in PDF output, "Easy to use and easy to understand", "The PDF outcome of the system could be more sorted, e.g. in a table view", "This system can shorten work processes"	Usefulness: 3.8 Ease of use: 3.86 Satisfaction: 3.63

Table 8: Experiment results of test person #2

of the difficulties with the use of the PDF document, test person #2 could complete all steps in the automated testing relatively quickly and with only few errors. She stated that Design-Testing Framework does not meet her needs at the moment because the design testing is not common in AppFactory now, but it could be very helpful in the future. The automation of the design testing process could lead to more frequent and consistent design tests in the quality management department.

Test results of test person #3 are represented in table 9. Test person #3 had the least experience with software testing. He studies information management and is currently working in the customer management department of AppFactory. Initially, he had some problems using ALM and DesignTesting Framework but could understand both tools after a short time. Like the other two participants he made more errors testing the applications manually. Thus, he made 6 errors in Agree and 10 errors in Konzernkalender in the manual testing and respectively 2 errors using the tool. At the beginning he said that it is not possible at all to define the concrete values of the elements, but then he tried to do it using Paint and Internet. For the test of the first application with both methods he needed almost the same time – 22 minutes for manual testing and 21 minutes with DesignTesting Framework. The reason for this is that he searched for the results in the output document for a very long time. The testing of the second application took 3 minutes more with the use of the automated tool. Test person #3 had the lowest rating in the questionnaire about the usefulness of the system, probably because he is not familiar with software testing and does not need such a design testing tool in his work. But he can imagine that this tool could be very useful.

Test person	Introduction questions	Time	Number or errors	Observation, comments	Usability questions
#3	Male, 25-34 years old, Bachelor in information management, internship in customer management, uses computer and mobile devices every day, has experience in testing	Manual testing: iAgree: 22 min KK: 8 min Automated testing: iAgree: 21 min KK: 11 min	Manual testing: iAgree: 6 errors KK: 10 errors Automated testing: iAgree: 2 errors KK: 2 errors	Manual testing: Used Paint to define attributes, had difficulties with defining the attributes, "It is not possible to define the values manually" Automated testing: Problems with finding elements in PDF output, the system can improve the effectiveness and reduce the time	Usefulness: 3.6 Ease of use: 3.43 Satisfaction: 3.38

Table 9: Experiment results of test person #3

6.6. Summary of the results

Analyzing the results of all three participants, it is obvious that all of them made more errors testing the design manually. It is not possible to find out the exact values with a human eye, so even using different helping methods they could not define the attributes precisely. It is particularly hard to define the sizes of the elements and font sizes, since they could look different because of the scalability of the screen. Also the colors can be perceived differently depending on the background color. The same color on the light background looks darker then on the dark background. Most of the few errors made during the automated testing were made because of the inattentiveness of the participants, because the results in the PDF document were right. The only deficiency in the system was in defining the width of the logotype. In Konzernkalender, the logotype had visually the same size as iAgree, but the tool got the width of 300 pixels compared with 121 pixels. That it because the original asset was differently cropped and the logotype image in Konzernkalender had white spacing on the sides, which is not visible on the application's screen. This task was marked as failed by all three participants because it was not possible to define the real size of the image with the tool. So the deficiency in the Design-testing Framework is that in some single cases the real values of the elements do not correspond to attributes seen on the screen.

It can be seen that the manual testing of iAgree always took more time than the automated testing of this application. Nevertheless, the design testing of Konzernkalender took more time with the use of the automated tool; however, this difference is small. The second application has fewer steps, so I can assume that the use of Design-Testing Framework can take the same or more time for completing the small tasks. The execution of the code and saving the result document at the beginning always takes some time, but evaluating the results after that can be done faster. That's why it is more efficient to use Design-Testing Framework for completing the long test cases at once. In general, all three

participants together spent 97 minutes for the manual testing, whereas using the automated tool they needed 10 minutes less. Assuming they test the design of mobile applications the whole working day of 8 hours, they would save 50 minutes per day.

According to the results of the questionnaire, all three test persons agreed that DesignTesting Framework is easy to use (4.67), can be useful in the testing tasks (4.33), can improve productivity (4.33) and can enable one to finish the job more quickly (4.33). The tool does not meet the needs of all test persons, since test person #3 does not work with software testing at all, and test person #2 does not perform design testing, but they all believe that DesignTesting Framework can improve job performance and would recommend it to others for design testing tasks (4.0). All answers given in the questionnaire got 3 points on the likert scale (neither agree nor disagree) or more (agree or strongly agree). That means that the system produced no negative impressions and all participants could imagine using it in testing work. The only question that got less than 3 points on the scale was: “I could use the system without instructions” (2.67). The participants said they would not know that they need to shake the device and where and how the result will be saved, but they could understand how to use the tool with the instructions very well. It is important to mention that test persons #1 and #2, who have more experience with the testing of applications, also had the higher rating in the questionnaire. Test person#3, with the least experience, gave the questions about the usefulness, ease of use and satisfaction a slightly lower rating. The DesignTesting Framework has documentation available and the experienced users can operate the system immediately after reading the instructions. Inexperienced users need some time to learn the tool, after which they can also use it very well. Overall the participants were satisfied with the design-testing tool (4.33); however, they would expect more additional functionality from the system (3.33).

The main problem mentioned by the participants was the representation of the results in the output document. All three test persons had difficulties in finding some elements in the PDF document because the keywords were not clear enough and the order of the elements was free. In addition, they felt irritated about different values of customer and corporate design requirements and did not know which value was asked. So better naming of the elements in the design process of the interface and greater differentiation between customer and corporate design requirements is needed. Solving this problem could avoid errors resulting from inattentiveness, reduce the time of the test, and as a result increase the effectiveness of the tool even more.

The analysis of the results of this research makes it possible to confirm that the target group of the experiment – people working with software testing or with the creation of customer requirements can easily use Design-Testing Framework for the design testing tasks, and find it very useful and efficient. The automated tool can reduce the time of the testing process and increase the accuracy of the results, although some improvements in the visual representation of the outcome are required.

7. Conclusion

Because of the rapid increase in the usage of mobile applications and establishing them in all areas, from entertainment to business, the topic of quality management and the testing of handheld applications has become a relevant research field. In order to reduce the testing time and to increase productivity of the application's production, the automation of the testing process is needed. The automatic capture, analysis and critique of mobile applications can simulate all possible user actions and repeat them multiple times, something which will help to save time and to cover the wide range of test cases. It is able not only to find automatically the problems and bugs of the system, but can analyze them and suggest possible solutions.

Since the need for automated testing tools is evident, a lot of different techniques for testing mobile applications have been presented in recent years. This thesis has given an overview on the state of the art in the development of automated GUI testing tools. Most of the presented techniques, such as Android Instrumentation, Robotium, MonkeyRunner, apktool, iOS Instruments, Hierarchy Viewer, and others, are based on the creation of a hierarchical structure of the GUI and analyzing the UI elements and their connections. Another part of the existing automated testing tools, such as Sikuli and eggPlant, are based on the image recognition technique through the capture of the application's screen. However, all presented tools evaluate the functionality and usability of the application, including how the system responds to user interaction with the UI elements. No tools for the testing of the visual appearance and corporate design of the mobile application have been found. Since testing of design guidelines and corporate identity is also an important issue, especially in the business context, the main objective of this thesis was to introduce the automated testing tool for the evaluation of customer and corporate design requirements within a company environment.

In this thesis I have proposed different ideas for the automated design testing tool, based on existing techniques: source code analysis of the layout files, source code analysis of the application code, screenshot analysis through the image recognition tools, screenshot analysis through image comparison, and combination of various methods. Finally, I have implemented the prototype for the dynamic source code analysis of the application code, since this technique can be executed faster and can define the attributes of the UI elements more precisely. Furthermore, I have developed the design guidelines for mobile devices, conforming to their features and limitations, based on the physical constraints of handheld devices. These guidelines can be used for the automated, as well as for the manual testing, of the application design.

The outcome of this work is DesignTesting Framework for testing the design requirements of iOS applications, according to the developed guidelines, the corporate design of the brand and the special needs of the customer. It is implemented in Objective-C and can be linked directly to the Xcode project of the required application. The Design-Testing Framework can be activated with the shaking movement during the runtime of the application. A tool reads the design requirement documents, searches for all UI elements of the current screen and proves their attributes, such typefaces, colors, sizes and spacing, according to the values in the documents. The results of the test are saved in the structured PDF document as the text output. The DesignTesting Framework provides the following functionality: working with any iOS application with the available source code, orientation and resolution awareness, setting of user preferences, storing data with iTunes,

finding all UI elements of the current view, defining the certain attributes of required elements, reading the design requirements from the excel document, comparing the UI element attributes with the customer and corporate design requirements, analyzing the general design requirements, comparing the screenshots of the UI elements with the assets and result output. These functions are implemented in two classes – DesignTestingViewController class, which contains the core methods for the execution of the framework, finding the UI elements and rendering the output document, and VWCVSAttributes class, which defines the attributes, reads the excel requirement documents and compares the attributes. My automated design-testing tool gives the opportunity to verify the attributes of the UI components, which cannot be recognized by the human.

The prototype was tested with two iOS applications – iAgree and Konzernkalender, and returned correct results for both of them. My automated tool was evaluated in the research experiment, where 3 users tested the design of two applications, both manually and with the use of DesignTesting Framework. They had to complete four test cases with numerous steps, record the results in ALM and fill out the usability questionnaire. During the user study, qualitative and quantitative data were collected, including time, number of errors and satisfaction with the tool. The results from the study show that the tool was understandable, easy to use, and that the testers found it useful. They think it can reduce the time and increase the effectiveness of the design testing process. The tasks performed with the automated tool were done faster and almost without errors, while manual testing resulted in plenty of incorrect answers.

The evaluation of results suggests some improvements that can be done in a future work, including the recognition of additional tested components, the awareness of different ways of the application's implementation, identifying dynamic requirements, inventing the rules for designers and developers for the naming and description of the UI elements, possible combination of source code analysis with image recognition, and better visual representation of the test results. The improvement of these points in the future work will surely make the system more efficient, consume less time and effort, and bring more accurate results. The invention of the automated testing tool such as DesignTesting Framework in the design testing process will lead to the development of more qualitative and visually appealing applications and to a reduction in production costs.

8. Future Work

The analysis of the experiment results shows that the use of the automated testing tool had a successful outcome and can improve the productivity of the testing process. In addition, it meets the need of quality management, because no equivalent tool is currently known, and the design of the application is tested either manually or not tested at all. However, there are some problems and limitations that have not been implemented yet. Also the results of the research experiment showed that some issues could be still improved in future work.

The current version of the prototype tests the incomplete number of the core design requirements, such as typefaces, sizes, colors, alignments and spacing of the labels, images, buttons and table elements. The next version of the system should include the recognition of additional UI types to enable the evaluation of all possible elements, for example tab bars, navigation bars, headers, footers, borderlines, count indicators, sliders and switches. It should also differentiate between different types of sliders (vertical slider, horizontal slider or menu slider), images (icons or large pictures), text elements (title, subtitle, headline, normal text, text in the table cell, title of the table or links)and other components. On the one hand, the implementation of the system should include functions for identifying these UI elements and, on the other the Excel template should provide the option to choose different types of these elements, so that they can be correlated to the elements in the source code. In addition, further attributes can be proved by the system for more detailed results, for example the typeface weight of the labels (regular, italic or bold).

Since all applications are implemented in different ways, some standardization is needed. One possibility would be to distinguish in the source code of the tool between all potential manners of implementation and to offer the solutions for all of them. However, it is not always feasible to provide all possible ways and can make the code unnecessarily complex. Another possibility is to invent the rules in the storyboard for the developers, which defines how the GUI components must be described during the implementation. Additionally, the storyboard for designers, as well as for the developers, should include the rules for the designation of the elements in order to provide them with unique and self-descriptive names. According to the method of description of UI elements, the system must provide the functionality to identify only those elements that are currently visible on the screen. If the tool recognizes all elements that are implemented for this view, but not shown on the screen at the moment, it must define which of them are activated and which are hidden. It also should be able to recognize the objects in the pop-up window opened at the time of the testing. It can be done either through adding required functionality to the source code or through recognizing the visible elements via screen capture.

Furthermore, the dynamic requirements can be better controlled in the future version. A tool should enable operation with proportional values and the identifying of attributes dependent on other variables, such as screen resolution or scalability factor. It also must give the opportunity to work with constraints and to calculate the minimum or maximum sizes and spacing of the elements, dependent on their location to other components. For this, the system can use the detailed description of all dynamic directives in the requirement document and search for the required or closest elements, taking into account the coordinates of the objects. The use of image recognition can also help in this case, through finding the objects on the screen capture.

The combination of the source code analysis with the image recognition is in general a good approach for future work. It can bring different benefits. In some cases it may be not possible to find the closest element in order to calculate the spacing to it, because GUI of most applications consist of large number of subviews stacked into each other. Some subviews can function as containers to group the other elements inside of them, and are not visible on the screen. However, it is required to evaluate only visible objects. With source code analysis, it is difficult to define how certain elements are rendered on the screen. The image recognition of the screen capture considers the real representation of the elements that are actually visible in the current view. So the image recognition technique can be used to find all UI components on the screen and to define the closest elements in relation to the tested object. After that, the tool can search these objects in the source code through defining the coordinates on the screen or through comparing the screenshots of every element and to perform all needed operations with them programmatically. This technique can bring more accurate results, especially for the dynamic elements.

As the outcome of the research experiment showed, the main problem for users was the output representation of the testing results. The improved version of DesignTesting Framework should use more evident keywords in the resulting PDF document. These keywords come either from the source code of application or from the design requirements documents, and therefore are given by the designers or the developers. So the solution of this problem is defining the rules for the designation of the UI elements, something that has already been mentioned before. In addition, the results should be represented more clearly to enable the easier searching for the required elements. The output of the components can be sorted by their names, ids or types, and can illustrate each component with a screenshot image. In addition, the elements can be represented in a table that shows the names of the elements and their attributes. The results of the test would be saved in a corresponding table cell with a word “right” or “false”, so that the tester can promptly see the bugs in the application. The information about these attributes and their values can be written small below the result. Such tabular representation of the test outcome can give better overview of all UI elements and the results, and can reduce the time and effort in searching for the needed elements.

Another problem detected in the research experiment was the imprecise differentiation between customer and corporate design requirements. The tool provides the possibility to select in the user preferences settings whether the customer or the corporate design should be tested. After executing the code, it shows in the output document only the results of the selected option. In the case of the tester wanting to test both requirements, the tool outputs all results for each element. The customer and corporate design results may have different values, as the requirements not always correspond to each other. It can be irritating when the analyzing the test results. One solution would be to output the results in two different documents, so that the tester can read first the results that he personally finds more important. Another option is to define which corporate design requirements are more important and should cover the customer requirements. So, for example, the logotype must always have the same size and must be located in the same place. The spacing between the buttons and the height of the elements of the same type must always be the same. Some other attributes, such as colors and typefaces, can differ if the customer has special wishes. So the future version of the automated tool should decide for itself which results are more important for the design evaluation, and display only these.

All these improvements will help make the DesignTesting Framework more easy to operate, make it faster, reduce the time and effort of the testing process, lead to more accurate and correct testing results, and consequently establish automated design testing in quality management and produce better designed mobile applications.

9. Acknowledgments

I would like to thank my supervisor, Prof. Dr. Gabriel Zachmann, for continuous support during my work on the master thesis and giving me helpful advices, and Prof. Peter von Maydell, for the help in design questions. I also would like to thank my supervisors at Volkswagen Group AppFactory: Rainer Riekert, who introduced me to Objective-C and helped in practical arrangements, and Ingo Wolterstorff for the help in testing questions and giving me useful feedback. In addition, thanks go to my collegues – Jan Söhlke, who helped me with the technical questions, and information about design requirements and used applications –Martin Bonneberg and Jennifer Jane Poerner, who provided me with useful information about Volkswagen corporate design; Michel Malkowsky, Rouven Hernier, Tim Weschpatat and Sebastian Kruschwitz, who gave me the access to the applications used for the testing and helped with practical information about them. Special thanks go to the testers, who participated in my research study.

10. References

1. Acord, C. G. and Murphy, C. C.: Cross-Platform Mobile Application Development: A Pattern-Based Approach. Thesis, Montrey, California, March 2012
2. Amalfitano D., Fasolino A. R., Tramontana, P, De Carmine, S., Memon A. F.: Using GUI Ripping for Automated Testing of Android Applications. ASE'12, Essen, Germany, September 2012
3. Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., Imparato, G.: A Toolset for GUI Testing of Android Applications. 28th IEEE International Conference on Software Maintenance (ICSM), 2012
4. Andriychenko, V. and Lin Y.: Automatic Functionality and Stability Testing Through GUI of Handheld Devices. National Chiao Tung University, November, 2011
5. AppPerfect: GUI Testing.
   URL: http://www.appperfect.com/products/application-testing/app-test-gui-testing.html
6. Balbo, S.: Automatic Evaluation of User Interface Usability: Dream or Reality. Proceedings of QCHI 95, 1995
7. Developer Android: MonkeyRunner.
   URL: http://developer.android.com/tools/help/monkeyrunner_concepts.html
8. eggPlant. URL: http://www.testplant.com/eggplant/
9. Fontblog: Die visuelle Führung der Marke Volkswagen. December 2007
   URL: http://www.fontblog.de/die-visuelle-fuehrung-der-marke-volkswagen
10. GitHub: DCIntrospect.
    URL: https://github.com/domesticcatsoftware/DCIntrospect
11. Goldberg, R., Schmauder, H., Schmidt, B.: Automatisierte, quantitative Analyse von Android-Applikation-GUIs. Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, Februar 2013
12. Graham, D., Fewster, M.: Experiences of Test Automation: Case Studies of Software Test Automation. Pearson Education, Inc., January 2012
13. Hu, C., Neamtiu, I.: Automating GUI Testing for Android Applications. AST '11, Waikiki, Honolulu, HI, USA , May 2011
14. Hughes Systique Corporation: Test Automation Tools for Mobile Applications: A brief survey. 2013
15. Ind, N.: The Corporate Image: Strategies for Effective Identity Programmes. Kogan Page, GB, 1990
16. Infoq: Functional GUI Testing Automation Patterns. August 2013
    URL: http://www.infoq.com/articles/gui-automation-patterns
17. iOS Human Interface Guidelines. Apple Inc., October 2013
18. iPDFDev, URL: http://ipdfdev.com/about-me/
19. Ivory, M. Y., Hearst, M. A.: The State of the Art in Automating Usability Evaluation of User Interfaces. ACM Computing Surveys (CSUR), Vol. 33 Issue 4, New York, USA, December 2001
20. Jovanović, I.: Software Testing Methods and Techniques. Belgrade, May 2008
21. Mandel, T.: The Elements of User Interface Design. WILEY, 1997
22. Martelin, T.: Orientation Awareness in Declarative User Interface Languages for Mobile Devices: A Case Study and Evaluation. Master's Thesis, Espoo, June 1, 2010

23. McNamara, M. T. Y., Guan Tan, C., Massey, D. T.: System and Method for Automated Design Verification. October 2000
24. Mobile statistics,
URL: http://www.mobilestatistics.com/mobile-statistics/
25. Mobilethinking,
URL: http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/e#appusers
26. Morgado, I.C., Paiva , A. C. R., Faria, J. P.: Reverse Engineering of Graphical User Interfaces. The Sixth International Conference on Software Engineering Advances (ICSEA), 2011
27. Muccini, H., Di Francesco, A,. Esposito, P.: Software Testing of Mobile Applications: Challenges and Future Research Direction. Automation of Software Test (AST), 7th International Workshop on, 2012
28. Pope, G. M., Stone, J.F., Gregory, J. A.: Automated Software Testing System. August 1994
29. Roberts, P.W., Dowling, G.R.: Corporate reputation and sustained superior financial performance. Strategic Management Journal, Vol. 23 No. 12, 2002
30. Rountev, A., Yan, D.: Static Reference Analysis for GUI Objects in Android Software. CGO ’14, Orlando, FL, USA , February 2014
31. Schneiderman, B., Plaisant: Designing the User Interface: Strategies for Effective Human-Computer Interaction (5th Edition). Addison Wesley Pub Co Inc., February 2009
32. Schubiger, P.: Der Corporate-Design-Prozess in der Beratung am Beispiel eines neuentwickelten Simulationstools. HWZ Hochschule für Wirtschaft Zürich, Mai 2012
33. Shirazi, A. S., Henze, N., Schmidt, A., Goldberg, R., Schmidt, B. and Schmauder, H.: Insights into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps. EICS’13, London, United Kingdom, June 24–27, 2013
34. Stuart, H.: Exploring the corporate identity/corporate image interface: An empirical study of accountancy firms. Journal of Communication Management, Vol. 2 No. 4, Stewart Publications, 1998
35. Szydlowski, M., Egele, M., Kruegel, C., Vigna, G.: Challenges for Dynamic Analysis of iOS Applications. iNetSec'11 Proceedings of the 2011 IFIP WG 11.4 international conference on Open Problems in Network Security, Springer-Verlag Berlin, Heidelberg, 2012
36. Tschernuth, M., Lettner, M., Mayrhofer, R.: Evaluation of Descriptive User Interface Methodologies for Mobile Devices. Computer Aided Systems Theory – EUROCAST 2011, Springer-Verlag Berlin, Heidelberg, 2012
37. Technopedia,
URL: http://www.techopedia.com/definition/2953/mobile-application-mobile-app
38. Technopedia: Graphical User Interface Testing.
URL: http://www.techopedia.com/definition/29846/graphical-user-interface-testing-gui-testing
39. Testwarriors: Comparison Report: Sikuli Vs Eggplant.
URL: http://testwarriors.blogspot.de/2012/04/comparison-report-sikuli-vs-eggplant.html
40. Tutorialspoint: Software Testing Methods.
URL: http://www.tutorialspoint.com/software_testing/testing_methods.htm
41. Van den Bosch, A. L. M., de Jong , M. D. T., Elving, W. J. L.: How corporate visual identity supports reputation. Corporate Communications: An International Journal Vol. 10 No. 2, 2005

42. Veracode: Static Testing vs. Dymamic Testing.
URL: http://blog.veracode.com/2013/12/static-testing-vs-dynamic-testing/
43. Volkswagen AG Corporate Design Styleguide. Volkswagen Aktiengesellschaft, 2013
44. Volkswagen Corporate Design. Volkswagen, 2012
45. Volkswagens AppFactory: Die Produktion für Smartphone und Tablet brummt. September 2012, URL: http://www.it-region38.de/-/volkswagens-appfactory-die-produktion-fur-smartphone-und-tablet-brummt
46. Weinschenk, S., Yeo, S.C.: Guidelines for Enterprise-Wide GUI Design. 1995
47. Yeh, T., Chang, T. and Miller, R. C.:Sikuli: Using GUI screenshots for search and automation. UIST'09, Victoria, British Columbia, Canada, October, 2009
48. Yeh, T., Chang, T.-H., Miller, R. C.: Sikuli GUI Testing Using Computer Vision. CHI 2010, Atlanta, Georgia, USA , April 2010
49. Zhang, D., Adipat, B.: Challenges, Methodologies, and Issues in the Usability Testing of Mobile Applications. International Journal of Human-Computer Interaction, 18:3, 293-308, November 2009
50. Zheng, C., Zhu, S. Dai, D., Gu, G., Gong, X., Han, X. Zou, W.: SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications. SPSM'12, Raleigh, North Carolina, USA , October 2012