Christine Pichler, BSc

Software Test Automation in the Field of RFID

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Institute for Software Technology

Graz, May 2015

Abstract

Although time consuming and complex, testing is an important and frequently applied process to verify the quality of software. Manual software testing is error-prone and might be less efficient, especially in terms of regression tests. Nevertheless, test automation is also challenging and especially in combination with Radio Frequency Identification (RFID). This master's thesis elaborates how to automate the test processes for multi-layered, distributed software frameworks using RFID. Basing on the software detego® SURVEYOR from the Enso Detego GmbH, functional tests are applied on unit and integration levels. Traditional unit testing has been used to verify the web service and parts of the web application, whereas functional Graphical User Interface (GUI) tests constitutes the greater part for the front-end applications. These are a web and a desktop application, the latter one communicates with an RFID-printer. Thus, several testing frameworks have been evaluated for testing the GUI. In order to be capable to support an automatic test procedure of the desktop application interacting with an RFID-printer, a hardware simulation has been used. For the mobile device application neither unit nor GUI tests where applicable such that an integration test has been developed on the device itself.

Keywords: distributed software, test automation, GUI test, graphical user interface test, unit test, integration test, RFID, Radio Frequency Identification

Abstract

Testen ist ein wichtiger, häufig verwendeter Prozess um die Qualität einer Software zu überprüfen, obwohl dieser zugleich zeitintensiv und komplex ist. Manuelles testen der Software ist dabei fehleranfällig und möglicherweise weniger effizient, vor allem in Hinblick auf Regressionstests. Nichtsdestotrotz bietet automatisiertes Testen genauso Herausforderungen, noch dazu in Kombination mit Radio Frequency Identification (RFID). Diese Masterarbeit beschäftigt sich mit der Automatisierung des Testprozesses für ein verteiltes Software System, welches RFID nutzt. Basierend auf der Software detego® SURVEYOR der Enso Detego GmbH wurden funktionale Tests auf Komponenten- und Integrationsebene angewendet. Dabei wurden traditionelle Komponententests zur Verifikation des Webservices und Teilen der Webapplikation verwendet. Funktionale Graphical User Interface (GUI)-Tests bildeten hingegen den größeren Anteil der Applikationen mit graphischer Benutzeroberfläche. Diese sind einerseits die Webapplikation, andererseits die Desktopapplikation, welche auch mit einem RFID-Drucker kommuniziert. Zum Testen der Oberfläche wurden verschiedene Frameworks evaluiert. Um einen automatisierten Testdurchlauf der Desktopapplikation, interagierend mit dem RFID-Drucker, durchführen zu können, wurde eine Hardware-Simulation genutzt. Die Applikation für das mobile Gerät konnte weder mit Komponenten- noch GUI-Tests verifiziert werden, daher wurde ein Integrationstest auf dem Gerät selbst entwickelt.

AFFIDAVIT

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

Date	Signature

Acknowledgements

First, I would like to thank my supervisor Franz Wotawa for his helpful guidance during this master's thesis, especially his patience and constructive criticism.

I would also like to thank Matthias Weitlaner for offering me the opportunity to undertake this academic work in cooperation with the Enso Detego GmbH. I would like to express my sincere gratitude to the Enso Detego staff – especially the product development team – for their support, the insightful discussions and for being amazing colleagues who make it a pleasure to work with. Additionally, I want to thank Michael Goller for proofreading this master's thesis and his helpful comments.

I would like to thank my cousin Conny for her invaluable advice and help, not only during this master's thesis but throughout my whole life. I would also like to extend my thanks to Doris, Manu, Nina and all my other friends for their friendship and for showing so much understanding and patience during the last couple of months.

Thanks to my parents Brigitte and Karl, and to my brother Dom, for their unconditional and perpetual support. I would not be the person I am today without them.

Last but not least I want to thank my cats Cleo and Kitty for their calm, except at 5am when they turn into the most effective and perseverant alarm clock in the world!

Contents

Chapter 1

Introduction

This master's thesis concerns automating a software testing process within the context of multi-layered, distributed software frameworks in the space of supply chain and article management using Radio Frequency Identification (RFID). Beyond the advantages of automation within software testing there are also some challenges to overcome.

1.1 Motivation

Software testing is expensive. About half of the time and money spent on the development process amounts to testing and debugging (Fraser, 2007; Myers et al., 2011, Introduction). Whenever the testing procedure is performed manually those costs may increase up to 80% (Burnim and Koushik, 2008). At the same time, testing is an important step within the software development process. The more complex a system becomes the more possibilities of failures and side effects arise. Those failures range from irrelevant, cosmetical issues up to serious risks in worst case, if the application is applied for example within a medical process or the aircraft technology. One well-known example is the failure of the radiation therapy machine *Therac-25* in the 1980s, which leaded to six known massive overdoses of radiation and caused deaths and serious injuries (Leveson and Turner, 1993).

Testing is used to strengthen the confidence in the developed software, such that the requirements are fulfilled and the software product does exactly what it is supposed to do. At least theoretically, the correctness of any system can be proven by a complete test run. The test suite, which is a collection of tests, must consider every possible case a system can run into to fulfill this purpose. In practice, even for small systems, the number of those cases becomes too large to be traceable. Sometimes formal proofs are suggested as alternatives to show the program's model correctness. Unfortunately, praxis shows that this kind of proof is equally not applicable for huge, scalable systems because formal proofs are too abstract (Fraser, 2007; Myers et al., 2011, Chapter 2).

As a consequence to the problems mentioned above, testing is inevitable. Although testing is complicated, in several companies the testing processes are still performed manually by humans. This leads to high implementation costs and bears additional challenges and problems (Collins et al., 2012). In general, complicated processes performed by humans are error-prone and tend to be incomplete (Fraser, 2007). In order to reduce these problems and the costs test procedures can be automated. This leads to an increased efficiency, especially when tests are repeated periodically.

Nevertheless, test automation is not simple either and results in other problems (Fraser, 2007). Decision problems occur, for example when selecting which tests belong to the test suite. This selection is needed, since exhaustive testing is impossible due to the enormous and sometimes even infinite number of possible tests. In addition to that, every executed test has to be evaluated, whether it detected a fault or not. In terms of efficiency, these issues should be faced automatically as well.

1.2 Problem Statement

This master's thesis aims to automate parts of the testing process for an existing software solution, namely *detego® SURVEYOR*, developed by the Enso Detego GmbH. The comprehensive platform provides configurable and personalized solutions with the help of RFID in the area of asset management and product life cycle management applications. RFID is, like barcodes or SmartCards, an Automatic Identification (Auto-ID) system (Finkenzeller, 2010). A further description of the technology and advantages of RFID can be found in Section 1.2.1.

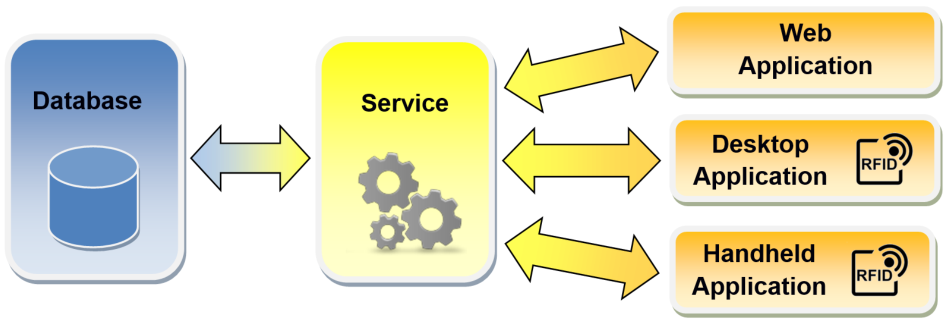


Figure 1.1: Overview of detego SURVEYOR, developed by Enso Detego GmbH

An overview about detego® SURVEYOR is provided in Figure 1.1. The software consists of a database as back-end and one or more applications in the front-end, with a dedicated interface service between the applications and the database.

In particular, the applications in the front-end are either web-based or rich client applications running on traditional PCs or mobile devices with Microsoft based operating systems. The applications consist of several modules with different functions for particular use cases in item-level based asset and item management. Most of those applications are connected to one or more RFID devices to interact with RFID transponders (read and/or write data). Currently, the web application do not perform any direct interaction with the RFID technology and is used for administration tasks and reporting.

A detailed description of detego® SURVEYOR can be found in Chapter 3, Section 3.1.

The RFID integration complicates the process of any test automation for the software additionally. Therefore, apart from a few basics in the back-end, the detego® SURVEYOR software is mainly tested in manual processes. In order to highlight the challenges resulting from the multi-layered, heterogeneous system components faced within this master's thesis some theoretical background is provided in the following. The next subsection describes RFID in more detail, before the challenges are further described.

1.2.1 RFID - Radio Frequency Identification

RFID systems are used to provide information about living beings and objects (Tamm and Tribowski, 2010, Chapter 1.1). RFID is a contactless technology which carries information by radio waves and is used for identification (Finkenzeller, 2010, Chapter 1). There are numerous advantages in comparison to other Auto-ID systems. Although the technology is similar to those of SmartCards, the main advantage is a contactless machine readability which does not even need visual contact. Additionally, dirt and optical covering have only minor influence on the readability. The read range differs from a few centimeters up to several meters, depending on the conditions and the used frequency band. Furthermore the reading speed is very fast and supports simultaneous reads and higher data-rates compared to barcode technologies.

As shown in Figure 1.2 every RFID system consists of two components - the *transponder* and the *reader*.

The ***reader*** or interrogator is a read or write/read device, depending on the design and technology (Finkenzeller, 2010, Chapter 1). Within this master's thesis it is not distinguished between these and the term *reader* refers to both types of data capture devices. In general, a reader has a radio frequency module (transmitter and receiver), a control unit and a coupling element (for example an antenna). Additionally, many readers contain an interface (serial, ethernet, bluetooth, USB or others) to transmit the data to a host system.

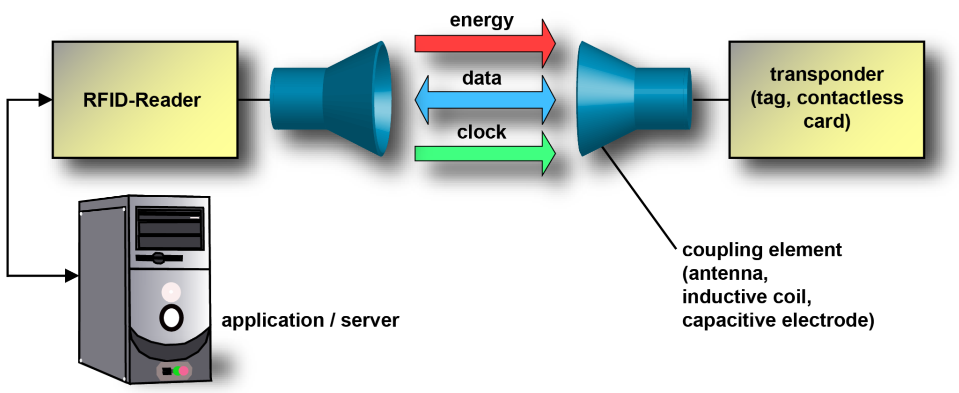

energy
data
clock
RFID-Reader
transponder (tag, contactless card)
application / server
coupling element (antenna, inductive coil, capacitive electrode)

Figure 1.2: Components of an RFID system (taken from the RFID Handbook[1])

The RFID readers existing on the market are similar but differ in ”the type of coupling (inductive - electromagnetic), the communication sequence (FDX, HDX, SEQ), the data transmission procedure from the transponder to the reader (load modulation, backscatter, subharmonic) and, last but not least, the frequency range” (Finkenzeller, 2010, Chapter 11). FDX means full-duplex, HDX half-duplex and SEQ sequential in this context. Moreover, the sizes range from small portable to large industrial reader. The antennas are typically connected to the reader using coaxial cables and mainly determine the read-range and read-performance. Furthermore, stationary readers typically have multiple antenna ports which are activated sequentially, one at the time. Thus, the (electro-) magnetic field of the reader can be enlarged with them (Tamm and Tribowski, 2010, Chapter 2). Figure 1.3 shows some antennas at the top, three stationary readers with up to eight antenna connectors at the left side next to two stationary readers with integrated antennas and several mobile devices at the right side.

RFID systems can be categorized by the operating frequency range. The most common ranges that are used are Low Frequency (LF), High Frequency (HF), Ultra High Frequency (UHF) and Super High Frequency (SHF). For different applications the one or the other frequency may be more practicable.

The ***transponder***, also called tag, is the *data-carrying* device (Finkenzeller, 2010, Chapter 1), which is typically attached to objects or living beings for the purpose of identification. Every transponder consists of a coupling element (i.e. an antenna or coil) and an electronic microchip where the data are stored. A selection of different transponders is shown in Figure 1.4. Transponders can be grouped according to their functionality (Finkenzeller, 2010, Chapter 2) (Tamm and Tribowski, 2010, Chapter 2):

- A *passive* transponder does not have any energy supply and draws the

[1]Last retrieved on 30.04.2014 from http://rfid-handbook.de/about-rfid.html

required energy from the reader field. This energy is used for the data transfer from the reader to the transponder and vice versa. This implies that the transponder is not able to communicate when not being located in the reader's interrogation zone. In passive transponder systems, the link between reader and transponder is considered as the limiting factor within the communication. Transponder readability is influenced severely by the physical boundary conditions, the orientation between transponder and reader and the specific material properties of tagged objects. This introduces additional challenges for automated system testing, since it is difficult to perform reproducible test scenarios.

- An *active* transponder has its own power supply on it, for example a battery or a solar cell used for the microchip's operations and data transmissions. Since the transponder does not have to rely on the energy drawn from the reader field, communication ranges are usually significantly larger with active transponders.
- A *semi-active* transponder has an integrated energy supply like the active ones but do not use it for the data transmission from the transponder to the reader.

In this master's thesis the focus lies on passive transponders. Thus, the reader is the link between a transponder and a software application. All components operate together on the basis of the master-slave principle (Finkenzeller, 2010, Chapter 11). Therefore, the software application as master initiates every activity of the reader or transponder. The reader as slave receives the command and enters, now as master, into the communication with the transponder (slave). Figure 2.1 illustrates these relations.
Passive transponder only acts upon the reader's command and never independently. The only exception represents a simple read-only transponder which continuously transmit its own identification number whenever it enters the interrogation zone of a reader.

A simple read command from the application software may trigger several communication steps between the reader and a transponder (Finkenzeller, 2010, Chapter 11). The reader's main functions are the transponder activation, to structure the communication sequence with the transponder and to transfer data between the transponder and the application. The reader handles the low-level communication with the transponder in an autonomous way according to the specific protocol that is used in an application.

Another hardware within this context is the ***printer***. An RFID printer has the ability to print transponders optically and digitally. This means that the visual print occurs besides the programming of the transponder's identifier.

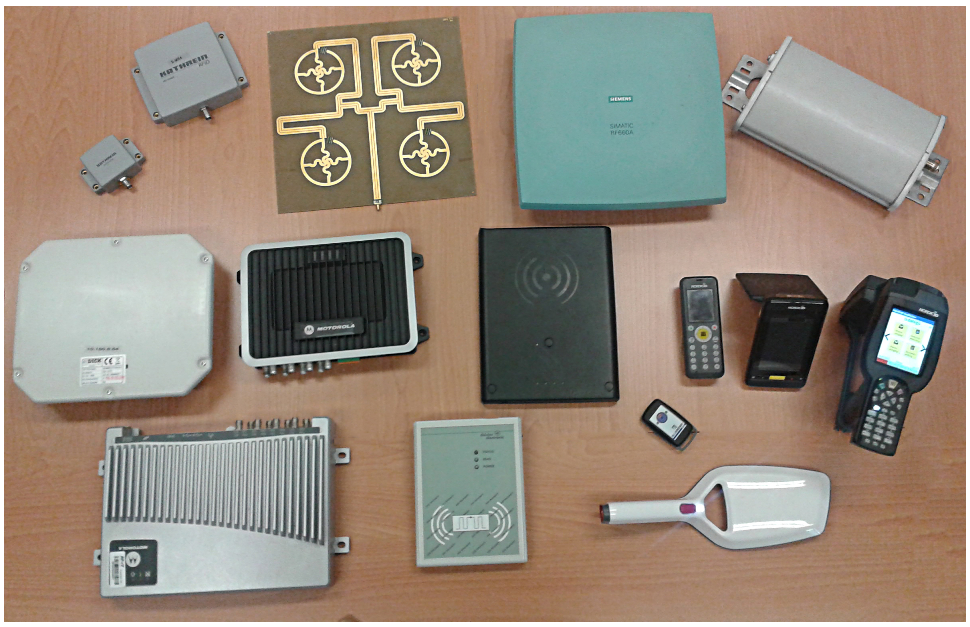


Figure 1.3: A selection of readers and antennas

1.2.2 Challenges

The first goal of this master's thesis is to set up a conceptual test run which evaluates how the different parts of detego® SURVEYOR can be tested in an effective way. In particular, the focus lies on automated testing and an investigation, which system components can be tested automatically. As described above, the test automation of any complex software application is difficult. There are even more challenges in this case, due to the peculiarities of RFID and because of the main parts of detego® SURVEYOR depend on interactions with RFID. According to the software it may make a difference in some points if none, one or more than one transponders are in the readers field. Therefore the processes which can be either tested

- fully automated,
- automated, with the help of hardware simulations or
- manually only

have to be identified first. Within this master's thesis the focus lies on functional tests. Nevertheless, the developed functional tests may be used for regression tests where applicable in the future. Depending on that knowledge, the appropriate test levels for specific test cases will be evaluated.

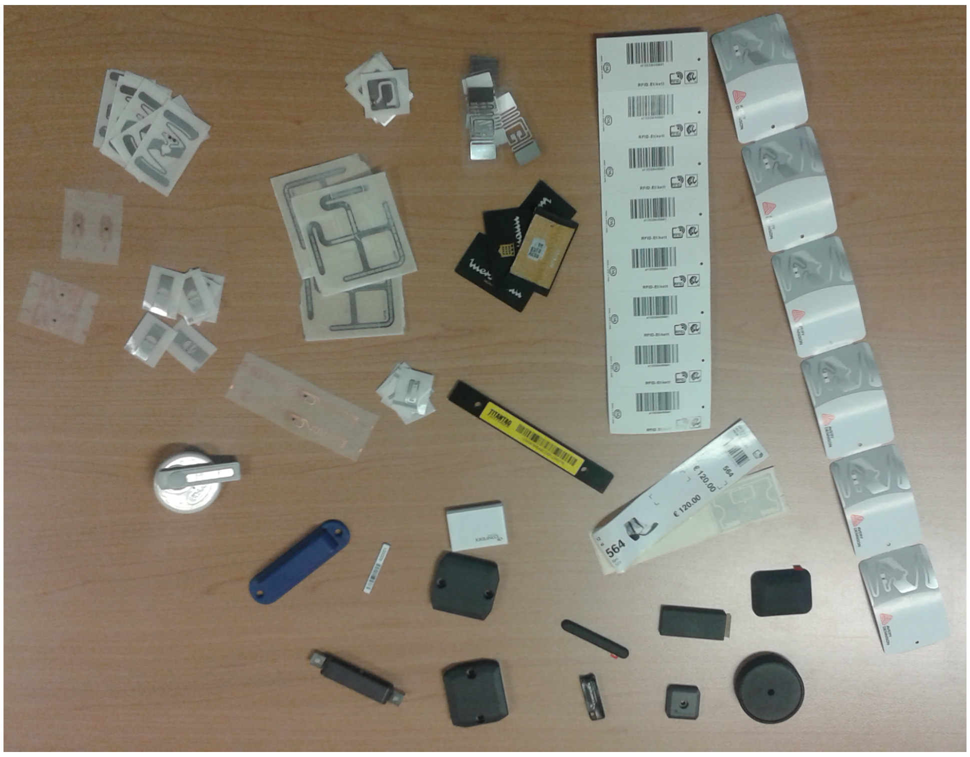


Figure 1.4: Different kind of transponders

Another challenge provides the web application. There are no RFID devices integrated and the design is built with Telerik's Kendo UI[2]. Especially for the reports, functional GUI tests would be helpful, because some data are calculated at the client's side. The problems with GUI tests are described in Chapter 2, Section 2.6. Special features of Telerik's Kendo UI may complicate them further.

The theoretical part of this thesis gives an overview of software testing in general, including testing levels, strategies and types. Furthermore, some details about the automation, including the advantages and challenges are mentioned as well as alternatives to software testing.
The practical part of this theses presents the implementation of automated tests within the detego® SURVEYOR platform. For this purpose, the implementation, Microsoft Microsoft Team Foundation Server (TFS) is used as collaboration and implementation framework.

[2]Last retrieved on 30.04.2015 from http://www.telerik.com/kendo-ui

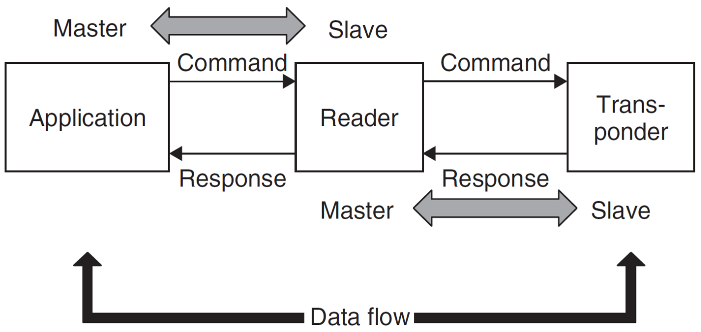


Figure 1.5: Master-slave principle within an RFID system (taken from Finkenzeller (2010, p. 318))

Chapter 2

Software Testing Preliminaries

The aim of every software developer is, among others, to develop a useful and robust software. The reliability and quality of software can be verified by testing, which continues to be the most important method for this. (Fraser et al., 2009)

2.1 What is Software Testing?

The history of software testing is as long as that of software development. At the beginning software testing had been a manageable process where test scenarios were written down on paper. With the development of the personal computer, software applications were subject to change and led to an increase of commercial software applications and concurrently to more competition. Consequently, the software test processes which nowadays map a huge number of combinations and permutations of test procedures changed too. (Dustin et al., 1999, Chapter 1)

There are several definitions for the process of software testing:

> ”Testing is the process of establishing confidence that a program or system does what it is supposed to.” (Hetzel, 1988, Chapter 1)
>
> ”Software testing is the process of analysing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.” (IEEE, 1990)
>
> ”Testing is the process of executing a program with the intent of finding errors.” (Myers et al., 2011, Chapter 2)

Summarizing these, testing aims to detect failures within software and to assure that the software does what it is supposed.

The frequently used terms error, fault, failure, bug or defect are sometimes used inconsistently in the literature. This master’s thesis adheres to the definition of Grindal and Lindström (2002), where a "***failure*** is defined as a deviation of the software from its expected delivery or service" and its cause is called ***fault***. The terms *error, bug* and *defect* are used as synonyms for *fault* within this document.

According to Dustin et al. (1999, Chapter 1) the earlier a failure is discovered, the less cost result during the development and test process. Table 2.1 gives an impression of the costs of a failure, detected in different stages of development.

Phase	Cost
Definition	$ 1
High-Level Design	$ 2
Low-Level Design	$ 5
Code	$ 10
Unit Test	$ 15
Integration Test	$ 22
System Test	$ 50
Post-Delivery	$ 100+

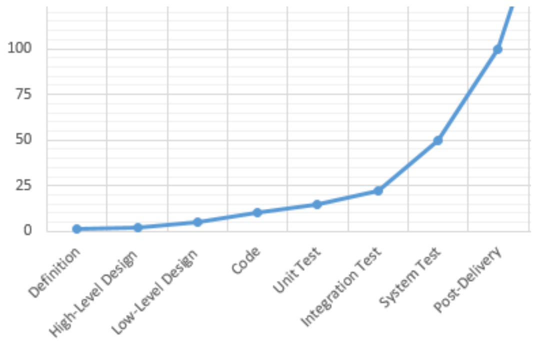


Table 2.1: Prevention is cheaper than cure (table taken from Dustin et al. (1999, Chapter 1))

Software testing is the process of verification and validation of particular software components. Both, verification and validation aim to detect as many errors as possible.

Validation refers to the software purpose and is usually performed on the complete software system. It answers the question if the implemented software is the right program with respect to intentions and wishes of customers. For this purpose, requirement documents specify the supposed purpose of the software. ***Verification***, in contrast, aims to investigate on the correct implementation of the software with respect to the design. This process can be performed on the entire software system or individual components, for example in early stages of development. Nevertheless a complete run of the verification process does not guarantee 100% of correctness (Fraser, 2007).

Today's software systems are typically characterized by complex, multi-layered architectures and consist of several components (Dustin et al., 1999, Chapter 1). Within typical client-server architecture, as sketched in Figure 2.1, the client-side is executed on different devices or applications, for example as desktop application, mobile application or web application. Consequently, the network link between the individual components is one critical aspect and therefore needs to be part of the testing and validation process.

Software testing can be performed in different ways, on either individual components or the whole system, from different points of view as well as with diverse methods. All these decisions depend on the software itself, as well as the purpose of the actual test case. The execution also may differ heavily, depending on whether it is performed manually or automatically. The following sections provide an overview of the testing processes, levels, strategies, methodologies and test automation.

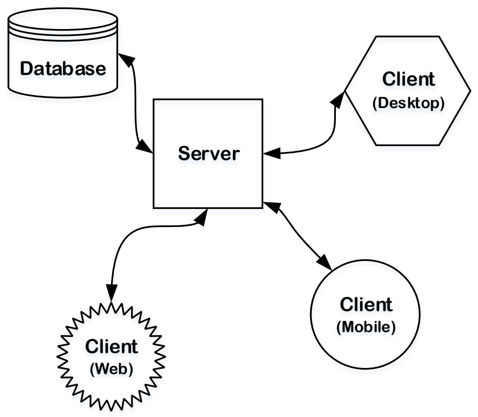


Figure 2.1: A possible client-server architecture

2.1.1 The Testing Team

The first and fundamental aspect within software development and testing processes is the organizational structure and the assignment of responsibilities among a team. Especially in small companies it might be easier if the project's developer is also responsible for its quality assurance. However there are good reasons to avoid such a situation. Actually, source code should not be written and tested by the same person. The developer is too intimately involved into the system in order to take an objective look at it. (Benetley et al., 2005) Additionally, according to Benetley et al. (2005) a good developer is not necessarily a good tester and vice versa. This does not imply that a tester does not need programming skills though according to Whittaker (2000) they do.
Consequently, software testers should build their own team within the software development process. The success - or failure - is widely influenced by the team's potential. Ideally, the team consists of several members with experience levels from beginners to experts which have a blend of technical and domain expertise and, depending on the subject matter, technology and testing techniques (Dustin, 2002, Chapter 3).

2.2 Testing Levels

During the development process, different testing levels can be identified which either belong to the verification or validation sector. The five testing levels visualized in Figure 2.2 - Unit, Integration, System, Acceptance and User-Interface testing - map the entire software testing process. Starting with tests on small and individual software components (units) towards complete integration tests (Pan, 1999; Software Testing Fundamentals; Myers et al., 2011, Chapter 5 and 6). Thereby, Unit, Integration and System tests are verification processes, whereas Acceptance and User-Interface tests belong to the validation sector.

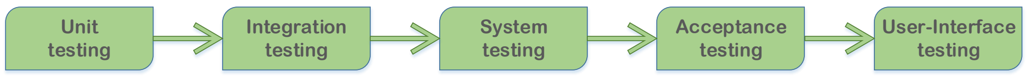


Figure 2.2: The five testing levels

- **Unit testing**
  This level of software testing refers to the smallest individual components, called units, of a software system. The purpose is to ensure that every piece of the software system works correctly based on a set of assumptions (Osherove, 2009, Chapter 1).

- **Integration testing**
  Integration tests are a combination of several unit tests depending on each other. The purpose of integration tests is hence to identify failures resulting from unit interaction. (Osherove, 2009, Chapter 1).

- **System testing**
  System testing means to test the complete software system within an integrated environment. The system is evaluated against the software specification and the specific functional requirements (Fraser, 2007).

- **Acceptance testing**
  This kind of software testing evaluates whether the system fulfils the business requirements and the current needs of the end-users (Myers et al., 2011, Chapter 6).

- **User-Interface testing**
  User-Interface (UI) tests aim to verify that the requirements are fulfilled and that non-functional aspects like usability or performance are met by the UI (Baker et al., 2008).

The definition of the term ***unit*** as smallest individual and testable component of a program is not consistent throughout the literature. Osherove (2009, Chapter 1) defined a unit as method or function, in contrast to Myers et al. (2011, Chapter 5), who refer to the individual sub-programs, subroutines, or procedures in a program. In Object-Oriented (OO) programs, a unit is defined as a method which normally belongs to any kind of abstract or derived class. When talking about procedural programs units typically refer to concerns to a module, although a module might in turn consist of several individual units (Software Testing Fundamentals).

According to Labiche et al. (2000) the categorization of Unit and Integration testing can not be separated in the same way for OO Software like for procedural software. Object operations can also be tested individually but in most cases the objects' behavior is more complex and depends on other objects.

2.3 Testing Strategies

According to Myers et al. (2011, Chapter 2) there are several testing strategies that differ in the initial situations as well as the execution and therefore yield in different results (Pan, 1999; Myers et al., 2011; Software Testing Fundamentals). The most commonly used strategies are:

- **Black-Box testing**
  In this context *Black-Box* means that the internal structure, design and source code is hidden from the tester. One can imagine a opaque box covering the program under test. Usually these are functional tests but do not necessarily need to be. This procedure is also called data-driven or input/output-driven testing.

- **White-Box testing**
  Contrary to Black-Box techniques, the tester knows the internal structure, design and source code of the program when using this strategy. The term *White-Box* is referred to a transparent box covering the source code. As a direct consequence, the tester needs to have programming skills in order to understand and evaluate the internal system structure. The input has to be specified depending on the software parts to test.

- **Gray-Box testing**
  A Gray-Box test is the combination of Black- and White-Box test methods. The tester knows the internal structure, design and source code while designing and creating the test cases. The test execution is performed from a user's point of view who does not know the software details. According to Dustin (2002, Chapter 4), Gray-Box testing is more efficient for companies than Black-Box tests.

2.4 Testing Types

Tests can be divided into different types depending under which aspect the software is tested. The different aspects may exclude each other partly, for example a high-performance system might not be as secure as possible and vice versa. In this case, humans have to decide which aspect is more important within the content to be proofed (Pan, 1999; Myers et al., 2011; Software Testing Fundamentals).

- **Functional tests**
  This test type refers to the reliability of a software program. The goal is a failure-free program as possible in terms that the application fulfils the requirements. Input (data, user interaction) is provided, executed and the received output is evaluated. Since functional tests are dominantly driven by an input-output evaluation, the most commonly used test method is Black-Box testing.

- **Security tests**
  Security is getting more and more important, especially for web-based applications. Security tests aim to identify and remove weak points in the software which can lead to violations. Another aspect is the analysis of security measurements effectiveness.

- **Performance tests**
  These tests assess a system's performance in reference to responsiveness and stability under certain conditions. The main term *performance tests* can be divided into several types. The four basic types thereof are:

  - **Stress tests**
  - **Load tests**
  - **Endurance tests**
  - **Spike tests**

- **Regression tests**
  Regression tests prevent a system from negative effects caused by software improvements and other changes which may occur in a systems lifetime. For this purpose, automated tests ()see Section 2.6) are useful to verify that already working parts of the software behave in the same manner after any changes.

Whereas functional and regression tests will be treated within the practical part, both, security and Performance tests are considered out of scope of this master's thesis.

2.5 Performing Tests

In praxis the test levels, strategies and types mentioned in the previous chapters operate together. However, the tester has to decide whether to use a random or a structural approach for the whole test procedure. Several studies had been conducted about the weaknesses and strengths of those in the past. (Duran and Ntafos, 1984; Gutjahr, 1999; Hamlet, 1994, 2006; Janhunen et al., 2011; Ntafos, 1998) According to Hamlet (1994), random tests are the most used and less useful ones at the same time.

An example are ***Monkey tests***, which are random tests based on model tests without any test specifications. They are executed in the way of Black-Box testing, hence the internal structure itself is not of interest hereby. They are usually operated at the level of system testing.

Agile tests are a special type of testing which is bound to the agile software development methodology. That include pair-programming, extreme-programming, test-driven development, among others. Hence, agile development and therefore agile testing as well, are dynamic processes, which are performed parallel to each other.

2.6 Test Automation

Test automation can be divided into different categories, reaching from test management to test execution and result interpretation. The two most important processes are the ***Test-case generation*** (Section 2.6.2) and the ***Test-execution***(Section 2.6.3). Whereas automated tools for test organization and execution are quite common nowadays, an automated creation of test-case generation is still not widespread (Fraser et al., 2009; Wissink and Amaro, 2006).

2.6.1 Why Automate Tests?

All kinds of tests mentioned above can - and traditionally are - executed manually. The majority of software intensive systems nowadays are indeed tested manually, irrespective of the components being new developments or established programs. This applies for both large-scale systems as well as individual and small projects (Wissink and Amaro, 2006).

Dustin et al. (1999, Summery of Chapter 1) state that

> ”the use of automated test tools to support the test process is prov-

ing to be beneficial in terms of product quality and minimizing project schedule and effort."

The manual development of test cases is tedious, time-consuming and error-prone (Fraser et al., 2009). Automation can improve this process significantly. Burnim and Koushik (2008) mention that typically the costs for manual testing amount to 50 - 80% of the total software development costs. According to Collins et al. (2012), automated tests increase the efficiency by saving time especially when test have to be executed repeatedly after any changes to the software, for example when performing regression tests. Besides, the sets of test cases are often more complete because of a systematically generation. Therefore, automation can be seen as an attempt to gain more and invest less.

Another reason to use test automation is the fact that in, among others, industrial applications the way of testing has to be systematic and traceable. This especially applies for software which acts in a safety-related system (Fraser et al., 2009).

Nevertheless, the test automation strategy of any software project or product has to be defined and analysed carefully. Depending on it, test automation might not be applicable or more expensive, whereas in other cases it can reduce software testing costs drastically. Consequential to benefit from automated tests, the tester has to be involved as early as possible in the development process. Besides, the testers have to know the testing tools and verification of test results in order to and compare the costs of automated and manual tests (Collins et al., 2012; Dustin et al., 1999, Chapter 2).

2.6.2 Test Case Generation

Software test research activities resulted in several different approaches to generate test cases automatically. Achieving a useful test run is important to find appropriate test cases. The generation can be either base on a modelling approach or may be derived directly from the software source code. (Boghdady et al., 2011).

- **Model-based**
  This is the most common test-case generation technique used today. Nevertheless, it is a challenge to define a formal model which can later be converted into a concrete set of test cases. Boghdady et al. (2011) categorize all different types of formal models into the main categories *requirements models, usage models* and *source code dependant models.*
  A non-deterministic model can lead to an *inconclusive* verdict because neither the pass nor the fail of the test can be concluded. This case may occur when the tests for non-deterministic model finally run on possible,

deterministic choices of the model. Consequently, when an executed test case fails because of a different but valid decision at the nondeterministic branching points, this can result in an erroneous fault detection and therefore an inconclusive test result (Fraser and Wotawa, 2007; Fraser et al., 2009).

- **Code-driven**
  Deriving test cases from the source code of the application started with *randomly-generated* test inputs approximately 30 years ago. This solution scales well but it is highly improbable that every possible situation is considered. Related techniques are *concolic testing* and comparable, which run simultaneously concrete and symbolic executions. A relatively young example is Microsofts automatic tool PEX for White-Box test generations (Burnim and Koushik, 2008; Tillmann and Halleux, 2008).

One has to keep in mind that generated test cases are abstract. To be able to execute such a generated test set, suitable test data are needed. Therefore, the test data selection or generation can be considered at least a problem, probably as important as the test-case generation (Fraser et al., 2009; Boghdady et al., 2011).

2.6.3 Test Execution

The next issue is a decision problem which is also known as oracle problem. Whether the test-case execution detected a fault or not has to be decided somehow. To reach a useful automated test execution, this issue and the test data generation should be treated with an automated mechanism too. (Fraser, 2007)

2.6.3.1 Graphical User Interface

Today, the GUI build an essential part of software. The way of interacting with computer programs has become much easier with the use of graphical user interfaces. At the same time, GUIs increase the complexity of any System Under Test (SUT) and present new challenges. Ruiz and Price (2007) list six issues in this context:

- GUIs are not designed for computer programs, such as automated tests, but for humans. Nevertheless, tests have to be automated.
- Traditional unit tests isolated in small components are not applicable for GUIs. In terms of GUI applications, a component can consist of more than one class and furthermore several components form a ***unit***.

- GUIs are driven by user-generated events. In order to test GUIs automatically, these events have to be simulated in any way. Later, the test program has to wait until the event has been transmitted to all listeners and the action's result are displayed to the user. Transforming such interactions to code can be a tedious and error-prone process.
- The number of interactions a person can perform within a GUI is high and there are a lot of potential paths from feature to feature, too. This increases the risk of developers introducing new bugs.
- Robust tests should not be affected by any changes of the GUI's layout.
- Test evaluations based on coverage code line criteria as used for conventional tests might not consider all user interaction scenarios. Besides of the tested code lines, the amount of possible states, testing each part of the application, is important too.

There are two common types of automated GUI testing. The first method is called ***record/playback***. Its main advantage is creating a test suite, which is a set of tests, in a short time span which can later be replayed by developers. The greatest disadvantages are expensive maintenance after any software changes and the need of existing GUIs to capture the user-generated events. After any changes in the GUI the recordings for according tests must be repeated.
The second possibility is to ***program*** GUI tests. This technique also supports test-driven development approaches where tests are written before the actual source code (Ruiz and Price, 2007).

Independent of the chosen type, GUI test automation faces several challenges and is considered as a highly complex domain. Jim Holmes[1] suggests some general principles to use this technique to bring a great benefit. Automating functions which do not change frequently, low-value features and third party functionality (for example the browsers HTML5 implementation) should be avoided. Using test infrastructure APIs minimize impacts of change by hiding complexity. Finally just high-value functional parts should be automated.

This leads to another approach namely ***functional GUI tests***. According to Zhu et al. (2008), this "means validating GUI objects, checking functional flows by operating GUI objects, and verifying output data which are generated in back-end and then displayed in front pages". Within this context a GUI object refers to an GUI element, such as a text, button, checkbox and so on. Contrary to traditional GUI tests these do not evaluate the look of the graphical interface but the software's behavior. The intent is to find failures within the software because the behavior after the interaction with GUI elements is not correct.

[1]Last retrieved on 1.05.2015 from http://blogs.telerik.com/kendoui/posts/13-06-19/four-ui-test-automation-tips-for-html5-applications

For example, if the GUI presents unexpected data after submitting a request by clicking a button.

2.6.4 Test Evaluation

Concluding the fact that software testing only detects, more or less successfully, existing bugs but does not show their absence, there is no decidedly end for testing. Theoretically, a program is tested to 100% if one or more tests have been executed for every possible case. If no more failures appear, the software is referred to as bug free. Unfortunately this is simply impossible in most cases. Hence the number of test possibilities is usually huge, sometimes even infinite. Whenever the testing process is profit-driven, the testing end occurs if either time, budget or the test cases are exhausted. Another rule to stop could be the reliability of the SUT or if the testing costs exceed the benefits (Fraser et al., 2009; Pan, 1999).

For this reason, one important aspect in sofware testing processes is to define economically and technically viable success criteria. At the same time, the SUT has to be reliable in order to meet the requirements. For this, the quality of test suites is measured. Two deterministic and commonly used methods are the *coverage analysis* and the *mutation analysis* (Fraser, 2007).

Coverage analysis concerns the parts of code which are covered by the tests. The result gives information about the amount of executed parts of the SUT while running the test suite. The common forms are based on the source code, but there are also others which analyse the lifetime of variables (data flow coverage), loops, race conditions, object code, formal models or even specifications. To name two common methods, the *statement coverage* reports on the lines of executed source code and *function coverage* depends on the executed functions to complete a test set.

Mutant analysis, in contrast, is used to evaluate the test set. Therefore, one has to create a set of mutants, either manual, or with a tool. A mutant is an altered version of the original software program where a fault, a so-called *mutation operator*, has been introduced, for example by changing one binary operator. The goal now is to determine if the test set can *kill* all mutants. A mutant is killed, i.e. detected, whenever at least one test case of the test set fails instead of passing. The success is measured by the *mutation score*. Due to numerous problems, especially the high computational costs, mutant analysis is not widely distributed in commercial use (Fraser et al., 2009; Mottu et al., 2006; Smith and Williams, 2009).

2.7 Alternatives and Supplements

What Dijkstra (1972) already stated in the 1970s still applies nowadays:

> "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

Actually, only about 50% of the errors within a program are found by a combination of unit, integration and system testing. Therefore, tests have a limit. One idea to deal with this issue is to use formal proofs. With a formal verification one can prove property violation or satisfaction. Unfortunately, this is mostly not sufficient in practice, since formal proofs do not include the environment (platform, compiler and so on) which directly influence the implementation. Precisely such a proof just shows that a given model fulfils a property. Furthermore, complex software programs can usually only be mapped to an abstract model. Consequently this technique does not avoid software testing, but those models help to derive test cases (Fraser et al., 2009).

Another approach are review techniques in order to read the code with the intention to find bugs. According to Dooley (2011, Chapter 15) those techniques can increase the percentage of found faults in the code to 93% - 99%. There are three important methods to review, namely code *walk-throughs*, *reviews* and *inspections*. They differ in the way of execution but are all based on the same principle *"Two heads are better than one"* and attempt to find failures. Depending on the amount of changes done within the code one or the other should be chosen. The best time to do the review is right after the first run of unit test set according to the new code part. In some cases it might be useful to do the review in a step between finishing the code and unit testing.

Code walk-throughs are the most informal review method. They apply for small bug fixes up to a few (not more than ten) lines of code. Typically, this method involves just the code author and one or two reviewers. The author explains the changes made to the code and their purpose to the reviewer. The reviewer should understand them and read the affected code. The reviewer either declines the change if there are any failures found, or accept it. In the case of detected failures the procedure repeats such that the author fixes the new found bugs and a new walk-through will be done afterwards. Once evaluated as failure-free, the code can be merged to the code base for further testing (Dooley, 2011, Chapter 15).

Code reviews are already more formal and designed for larger changes or new features within up to 500 lines of code. They are really meetings which involve between three and five participants with distinct perspectives. One of those is the code author, who moderates the review and takes notes. At least one of the attendees should be a developer who has a detailed knowledge about the project. As counterpart an experienced developer should also be present, who does not work on the same project to assume the quality perspective. The last part should be covered by a tester who considers the ways of testing the new code. Although the attempt is not to fix found failures within the review, the participants should go through the changed code and set up a list of found errors before the review takes place. This step is essential in order to make the review effective (Dooley, 2011, Chapter 15).

Code inspections are more expensive in terms of time and effort than the other two review techniques because a code inspection is not processed within one single meeting. (Dooley, 2011, Chapter 15) Therefore, they are mostly used in huge organizations. According to Eagan (1986), this process is separated into several phases and has distinct roles similar to code reviews. The phases are *planning*, giving an *overview*, *preparing*, the *inspection* itself, *rework* and *follow up*, where only the overview operation can be omitted in some cases. In contrast to code reviews, the author and the moderator are distinct persons, developers have either the task to paraphrase (the reader) or review (the reviewers) the code. Testers do not play a role here, but a recorder who takes notes and merges the failure list of the reviewers instead.

Chapter 3

Theoretical Concept

This chapter discusses in which manner the components of the software detego® SURVEYOR, which is used for the practical part of this master's thesis, may be tested.

3.1 detego SURVEYOR Details

This section describes the platform detego® SURVEYOR in more detail. The SURVEYOR platform provides intelligent article management functionalities tailored for the fashion retail industry. In particular, item management across the entire supply chain is implemented in softwar processes across the entire fashion supply chain.

The detego® SURVEYOR is a distributed system which can be separated into three main parts - a database, a web service component and several different applications. An architectural overview is shown in Figure 3.1, wherein the parts treated in this master's thesis are colored blue. The database builds the backbone and communicates with the clients (applications) via a web service. There are going to be some tests concerning the data provided by the database's views as well. Nevertheless, the database itself is not tested excessively and therefore it is not highlighted.

The detego® SURVEYOR platform is capable of dealing with complex business processes. For this reason, uniquely identified items (using the EPCglobal framework) are tracked within a state space model. The different states are illustrated in the workflow of Figure 3.2, where the possible states of an item are the circles and the rectangles describe different processes. In this example, the state machine covers the typical process in a retail store from goods inbound to the point of sale.

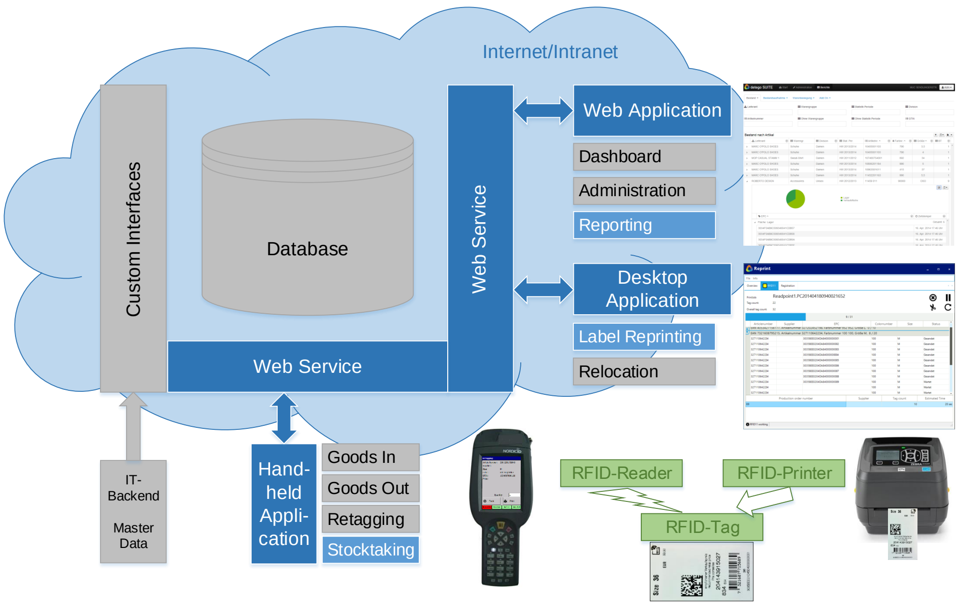


Figure 3.1: Architectural overview of detego SURVEYOR, developed by Enso Detego GmbH

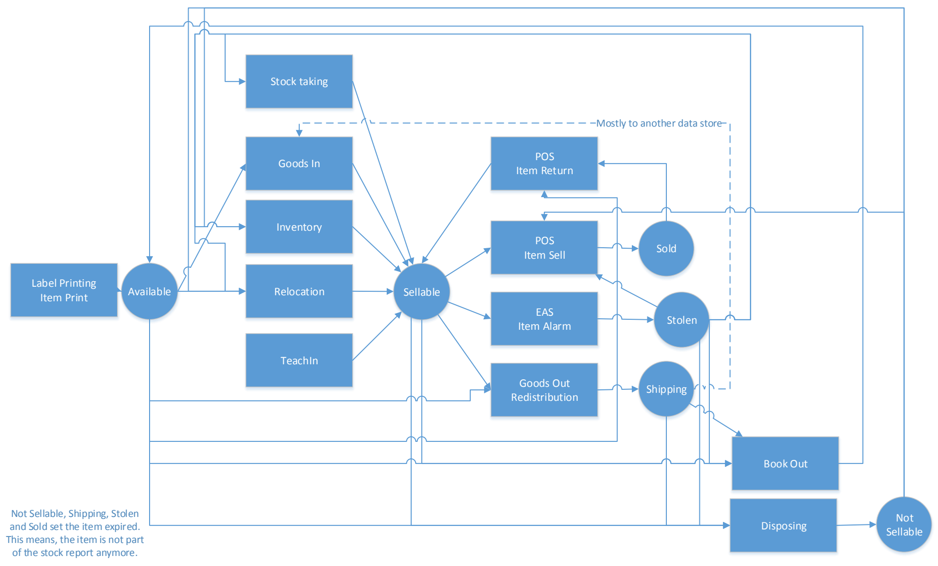

Stock taking
Mostly to another data store
Goods In
Inventory
Relocation
TeachIn
Label Printing Item Print
Available
Sellable
POS Item Return
POS Item Sell
EAS Item Alarm
Goods Out Redistribution
Sold
Stolen
Shipping
Book Out
Disposing
Not Sellable
Not Sellable, Shipping, Stolen and Sold set the item expired. This means, the item is not part of the stock report anymore.

Figure 3.2: Workflow of items, developed by Enso Detego GmbH

3.1.1 Database

The used database (detego® Data Store) is a relational Structured Query Language (SQL) database managed by a Microsoft SQL Server 2012. The relational programming language SQL is designed to communicate with a Database Management System (DBMS). Select, Insert, Update and Delete statements are used to query, define and manipulate the data. The origins of SQL date back to the early 1970s. (Sumathi and Esakkirajan, 2007)

Table 3.1 shows the most important entities of the detego® Data Store, which are used within this master's thesis.

Entity	**Description**
BusinessStep	Possible actions which may be performed on items (Initialize, Relocation...). Every item has exactly one business step, at a particular time instant.
Group	A collection of items. Used, for example, to define the item group performing a specific business steps.
Item	A unique entity with an Electronic Product Code (EPC). Every item represented by a transponder, is a product.
GroupItem	The mapping between groups and items. The amount of items of each product for each group is specified here too.
Product	The basis for every item. Used to define a product for which an arbitrary number of item instances may be available.
Site	The site defines a physical facility, such as a retail store.
State	Every item has a state, for example ”sellable” or ”inbound”.
Location	The location is a child entity of the site property. It defines areas of a site, i.e. sales floor or back room.
Readpoint	Used to identify the system component (i.e. PC or mobile device) where a software component is running on.

Table 3.1: Most relevant entities of detego Data Store

Most of the data needed by the applications is organized within database views. To retrieve these data, the applications are not allowed to access the database directly, but interact via a web service. The web service queries the database via the .NET entity framework[1], which is an object-relational mapper that allows developers to work with relational data using domain-specific objects.

[1]Last retrieved on 17.04.2015 from http://msdn.microsoft.com/en-us/data/aa937723

3.1.2 Web Service

By the definition of Booth et al. (2004) a web service is a software system, which supports machine-to-machine interaction over a network. Its interface can be , for example, described by the Web Services Description Language (WSDL), a machine-processable format, based on the Extensible Markup Language (XML). To interact with the web service, other systems use the Simple Object Access Protocol (SOAP), also based on XML, or the JavaScript Object Notation (JSON). Both types have advantages and restrictions. JSON uses the Open Data Protocol (OData), initially defined by Mircosoft, which "is a standardized protocol for creating and consuming data APIs"[2]. OData uses Hypertext Transfer Protocol (HTTP) as core protocol and accepts Representational State Transfer (REST) (Booth et al., 2004; Christensen et al., 2001).

> "The purpose of a Web service is to provide some functionality on behalf of its owner." (Booth et al., 2004)

Therefore, in the case of detego® SURVEYOR, the web service basically provides a well defined database-access layer in order to encapsulate critical functionality. To retrieve or modify data, the clients connect to the web service which than fetches the data from the database. This web service is provided by the Windows Communication Foundation (WCF) and hosted on Microsoft's Internet Information Service (IIS)[3]. WCF[4], a set of APIs, is part of the .NET framework to build service-oriented applications.
The web service of detego® SURVEYOR is written in C# and provides several interfaces for the applications.

The ***Authentication Service*** provides the authentication processes of any user within the detego® SURVEYOR environment. Some use cases are:

- Log in with username and password
- Change the password of an existing person
- Create a new login
- Log out

[2]Last retrieved on 17.04.2015 from http://www.odata.org/
[3]Last retrieved on 30.04.2015 from http://www.iis.net/
[4]Last retrieved on 30.04.2015 from http://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx

The ***Configuration Service*** provides the configuration processes of the modules within the applications of the detego® SURVEYOR. Some use cases are:

- Select the language (German, English) and load the according language strings
- Load specific application modules depending on the configuration
- Display service and reader status

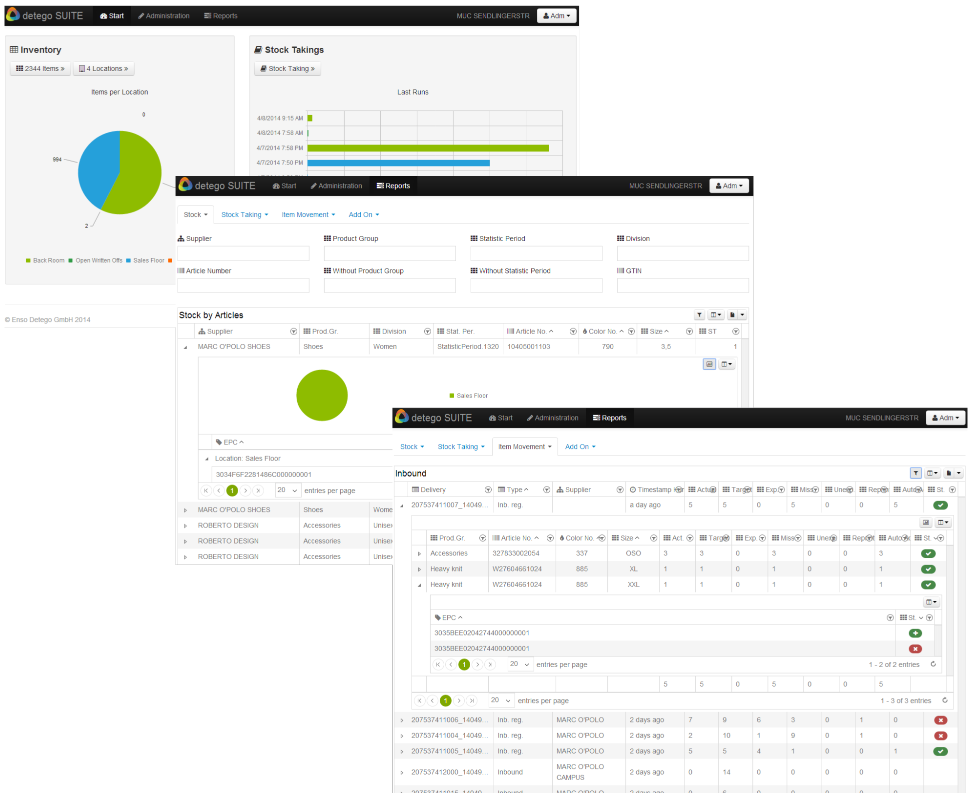
The ***Data Service*** and ***OData Service*** provide the data exchange between the applications and the database. The Data Service provides the functionality for SOAP calls, where every access to the database must be implemented in dedicated methods, which allows for more flexibility and an advanced business logic. The OData Service deals with JSON calls, where Select, Insert, Update and Delete statements are supported by default.

3.1.3 Applications

As already stated above, detego® SURVEYOR has several front-end applications for different purposes. Those which are relevant for this master's thesis are briefly explained in this subsection.

Web Application
The web application is used for administration and reporting purposes. A dashboard gives a quick overview about the stock level in a retail store, whereas the reports represent the whole stock information in detail, including the actual items' location, stock takings and performed item actions. In approximately 20 different reports the user can trace the movements of items and extract detailed stock information. To get an insight about the reports, have a look at Figure 3.3.
The main feature of the reports is their dynamic content. In this context dynamic means at one hand that actual data are retrieved on demand (open or refreshing the report) and on the other hand that the user can apply filters, show/hide columns, sort by column and open/close pre-grouped selections. Another feature is the PDF/CSV export of the reports' content. The administration area allows the user to perform special actions, for example write off individual items or configure modules and users. The displayed content within this area depends on the user's permissions.
It should be noted that the web application is not connected to any RFID device. The web application displays the data stored in the database and retrieved over the web service. Most of the functionality (i.e. manipulating the retrieved data by means of hiding and sorting the data, or providing the export of these and among others) is implemented using JavaScript (JS) on the client side. The

graphical visualization uses features of Kendo UI from Telerik[5] for displaying the data.



Stock by Articles

Supplier	Prod.Gr.	Division	Stat. Per.	Article No.	Color No.	Size	ST
MARC O'POLO SHOES	Shoes	Women	StatisticPeriod.1320	10405001103	790	3,5	1

EPC
Location: Sales Floor
3034F6F2281486C000000001

20 entries per page

Supplier	Prod.Gr.	Division
MARC O'POLO SHOES	Shoes	Wome
ROBERTO DESIGN	Accessories	Unise
ROBERTO DESIGN	Accessories	Unise
ROBERTO DESIGN	Accessories	Unise

Inbound

Delivery	Type	Supplier	Timestamp	Actu	Target	Exp	Miss	Unexp	Repr	Auto	St.
207537411007_14049...	Inb. reg.		a day ago	5	5	0	5	0	0	5	✓

Prod.Gr.	Article No.	Color No.	Size	Act.	Target	Exp.	Miss	Unexp.	Reprint	Auto Ac	St.
Accessories	327833002054	337	OSO	3	3	0	3	0	0	3	✓
Heavy knit	W27604661024	885	XL	1	1	0	1	0	0	1	✓
Heavy knit	W27604661024	885	XXL	1	1	0	1	0	0	1	✓
				5	5	0	5	0	0	5	

EPC	St.
3035BEE02042744000000001	+
3035BEE02042744000000001	✗

20 entries per page 1 - 2 of 2 entries

20 entries per page 1 - 3 of 3 entries

Delivery	Type	Supplier	Timestamp	Actu	Target	Exp	Miss	Unexp	Repr	Auto	St.
207537411006_14049...	Inb. reg.	MARC O'POLO	2 days ago	7	9	6	3	0	1	0	✗
207537411004_14049...	Inb. reg.	MARC O'POLO	2 days ago	2	10	1	9	0	1	0	✗
207537411005_14049...	Inb. reg.	MARC O'POLO	2 days ago	5	5	4	1	0	0	1	✓
207537412000_14049...	Inbound	MARC O'POLO CAMPUS	2 days ago	0	14	0	0	0	0	0	

Figure 3.3: Some screenshots of the dashboard and reports

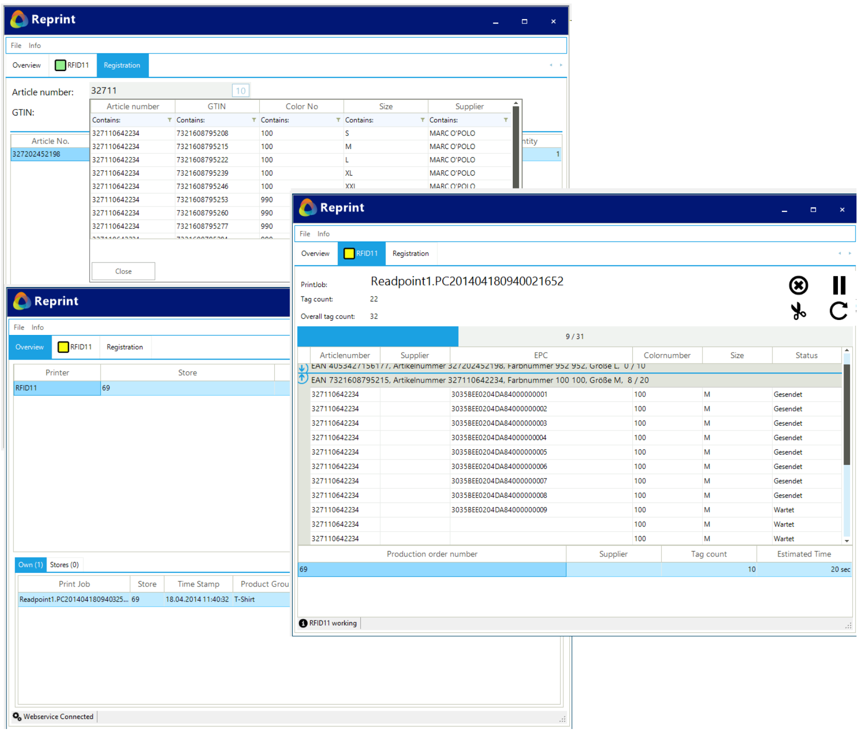
Desktop Application

The desktop application is a software installed on typical desktop PCs or workstations running on a Microsoft Windows Operating System (OS). Depending on the projects configuration, a desktop application contains a variety of modules that provide the desired functionality. This master's thesis focuses on one module named **label printing**. Some screenshots thereto are shown in Figure 3.4.

The application is connected to an RFID-printer, and is used to print new RFID-transponders, also called labels, if transponders are defect or lost. This process can be initiated by the mobile device application or within the label printing module itself. The user can define the product and the desired quantity for which labels shall be printed and has to start this prints actively by clicking a button. The system then prints and encodes RFID-labels with a specific optical design and layout and programs a unique identifier on every label. The GUI shows the current status of the printer and the web service (connected or not) and the print progress of the actual print. Furthermore, an amount of errors exists the application and/or printer may run into, and which must be handled

[5]Last retrieved on 17.04.2015 from http://www.telerik.com/kendo-ui

correctly. For example, like every printer, also an RFID-Printer may run into a paper jam or has no color ribbon left and others. The programming language here is C#.



Article number: 32711 10

GTIN:

Article number	GTIN	Color No	Size	Supplier
Contains:	Contains:	Contains:	Contains:	Contains:
327110642234	7321608795208	100	S	MARC O'POLO
327110642234	7321608795215	100	M	MARC O'POLO
327110642234	7321608795222	100	L	MARC O'POLO
327110642234	7321608795239	100	XL	MARC O'POLO
327110642234	7321608795246	100	XXL	MARC O'POLO
327110642234	7321608795253	990		
327110642234	7321608795260	990		
327110642234	7321608795277	990		

Article No.	ntity
327202452198	1

Printer	Store
RFID11	69

Print Job	Store	Time Stamp	Product Grou
Readpoint1.PC201404180940325...	69	18.04.2014 11:40:32	T-Shirt

PrintJob: Readpoint1.PC201404180940021652

Tag count: 22

Overall tag count: 32

9 / 31

Articlenumber	Supplier	EPC	Colornumber	Size	Status
EAN 4053427156177, Artikelnummer 327202452198, Farbnummer 952 952, Größe L, 0 / 10					
EAN 7321608795215, Artikelnummer 327110642234, Farbnummer 100 100, Größe M, 8 / 20					
327110642234		3035BEE0204DA84000000001	100	M	Gesendet
327110642234		3035BEE0204DA84000000002	100	M	Gesendet
327110642234		3035BEE0204DA84000000003	100	M	Gesendet
327110642234		3035BEE0204DA84000000004	100	M	Gesendet
327110642234		3035BEE0204DA84000000005	100	M	Gesendet
327110642234		3035BEE0204DA84000000006	100	M	Gesendet
327110642234		3035BEE0204DA84000000007	100	M	Gesendet
327110642234		3035BEE0204DA84000000008	100	M	Gesendet
327110642234		3035BEE0204DA84000000009	100	M	Wartet
327110642234			100	M	Wartet
327110642234			100	M	Wartet

Production order number	Supplier	Tag count	Estimated Time
69		10	20 sec

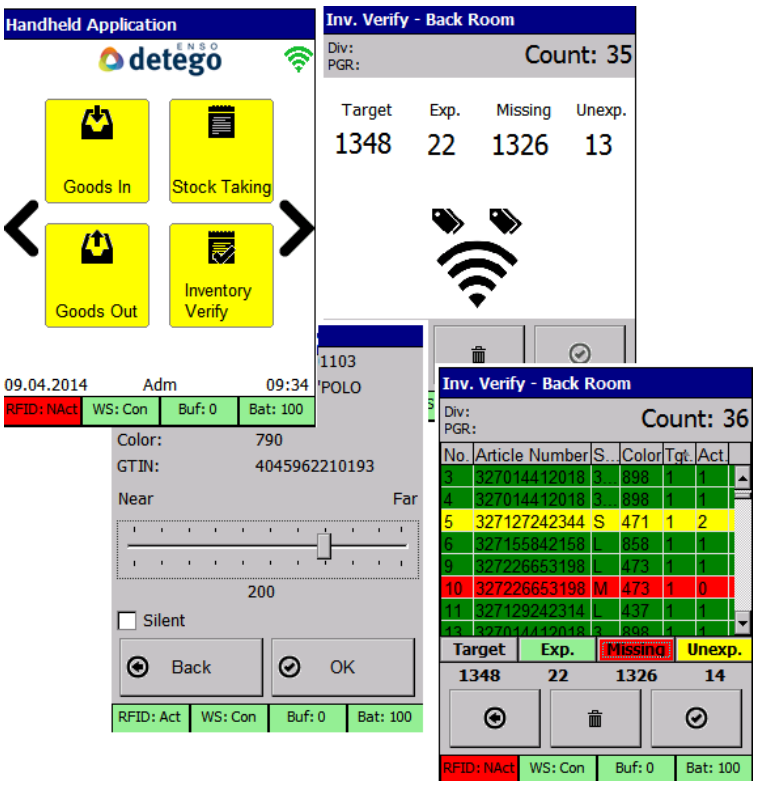
Figure 3.4: Printing a couple of labels

Mobile Device Application

Similarly, the mobile application consists of several modules and runs on top of a Windows CE driven, RFID enabled mobile device. These devices typically feature an RFID-reader and a barcode scanner and provide WIFI connectivity. For most of the features the mobile device has been preferred because of the portability, which has been a desired property. As already mentioned above, the modules inbound and label reprinting can trigger the reprint event of a transponder which is then handed over to the desktop application. Other features are the modules stock taking, article search, clarification of stock differences, write off and outbound. Again, this application is implemented in C#. Figure 3.5 shows some screenshots of the modules *inventory verification* and *article search*.

3.2 Possibilities to Test

There is a high number of test possibilities within the detego® SURVEYOR. The system consists of several components (applications, web service, database)


Handheld Application

09.04.2014 Adm 09:34

RFID: NAct | WS: Con | Buf: 0 | Bat: 100

Inv. Verify - Back Room

Div:
PGR:
Count: 35

Target	Exp.	Missing	Unexp.
1348	22	1326	13

1103
'POLO

Color: 790
GTIN: 4045962210193

Near Far

200

Silent

RFID: Act | WS: Con | Buf: 0 | Bat: 100

Inv. Verify - Back Room

Div:
PGR:
Count: 36

No.	Article Number	S...	Color	Tgt.	Act.
3	327014412018	3...	898	1	1
4	327014412018	3...	898	1	1
5	327127242344	S	471	1	2
6	327155842158	L	858	1	1
9	327226653198	L	473	1	1
10	327226653198	M	473	1	0
11	327129242314	L	437	1	1
13	327014412018	3	898	1	1

Target	Exp.	Missing	Unexp.
1348	22	1326	14

RFID: NAct | WS: Con | Buf: 0 | Bat: 100

Figure 3.5: Screenshots of different processes of the mobile device application

which interact with each other. The functional requirements of the system are specified in specification sheets. Therefore, it is a good point to start there because these functionalities have to be guaranteed. These use cases illustrate a sequence of events. To get an insight about the usual procedures, a selection of the most significant use cases is presented in the appendix A. Those do not cover the whole process nor all of the available modules and features. These use cases are taken as originals from the specification sheets of the Enso Detego GmbH. Hence, the project used as basis for this master's thesis has been designed for a German speaking company, the specification sheets are in German too. The order of the use cases is in an order, as they could be part of the life cycle of a label.

Other test cases arise from the experience with the applications. The reporting functionality is the most frequently used system component and thus the most important feature within the web application. It has to be verified that the right data are displayed after any changes. Hence, there are more than 20 different reports available, this lasts some time and even more if the verification has to be performed manually.

3.3 Actual State of Testing

3.3.1 Hardware

detego® SURVEYOR is capable to integrate several different RFID-devices, depending on the project's purpose. Within this master's thesis the following hardware is used:

- *RFID-Printer:* Zebra printer (desktop application)
- *RFID-Reader:* NordicID Merlin (mobile device application)

The hardware is bought in addition, such that the devices themselves are not tested by the Enso Detego GmbH. Of course they are tested as soon as they have been integrated within the software. For the integration an RFID-Middleware developed by Enso Detego GmbH is used. Within this middleware the RFID-Printer is tested with unit tests. The mobile device is not tested with unit tests. The reason for this is that testing the operations of the devices in an automatic way is challenging, hence they require one or more interactions, for example placing a transponder into the reader's field. Furthermore, the OS on the mobile device is still a Windows CE 6.0, which does not simplify the process either.

3.3.2 Software

Actually, the main part of the detego® SURVEYOR software is tested manually too. This includes all interactions with the applications. There have been trials with QUnit within the web application but due to time reasons, these have not been progressed further. However, the web service is tested with unit tests. Unfortunately, due to resource reasons these are neither complete nor maintained regularly. Hence, detego® SURVEYOR have been improved continuously in the past, parts of the unit tests fail and there is a need to adapt these. The actual code coverage of the web service amounts to about 40%.

3.4 Test Purposes

In general, the purpose within this master's thesis is to automate the suitable parts of the actual manual tests, such that the testing processes get more efficient. In this context *suitable parts* refers to those, which are cost-intensive to test manual at one hand, and relatively easy to automate at the other hand.

Web service
Hence, the web service is the central component of detego® SURVEYOR, bugs in this component are critical for the entire functionality. Therefore, it seems logical to test these functions thoroughly. Additionally, the service has no user interface, such that it could be tested manually. Consequently, the whole functionality has to be tested in an automated way and most effectively using unit tests. Hence, there is no user interface, GUI tests would not make any sense. For component or system tests encapsulated for the web service the same applies, because these are implicitly performed whenever these kinds of tests are executed on the applications' level.

Web application
The main usage and therefore also effort of test automation for the web application lies within the reporting section. There is also a dashboard and an administration part available, but their functionality is manageable. Nevertheless, tests for these parts might be automated in the future. Furthermore, the tests covered within this master's thesis will focus on the client-sided code. Parts of the functionality, provided within JavaScript code, will be tested using QUnit. This avoids for example cumbersome and cost-intensive tests concerning the displayed data and depending on the set filters. The data export function (PDF/CSV) is affected by the filters too, hence it exports the data represented within the web browser at the moment of the export.
Another aspect regarding automated tests of the web application is the data representation within the GUI (Kendo UI in this case) and the resulting features. This involves a data grid displaying the data retrieved from the service as well as the filter possibilities applied on the grid's columns. Testing of features, such that showing or hiding columns, showing data within pre-grouped levels for every grid as well as trying out every filter manually is time-intensive. All these functionalities could be tested as component tests with GUI tests. From now on, the term GUI tests refers to *functional GUI tests* (Chapter 2, Section 2.6.3.1).
The intent of these tests is to verify the displayed data before and after any user actions. Therefore, not every sequence of user interactions is tested but rather the main functionality. Implementing tests for each possibility of user input combinations is not feasible due to complexity reasons and the possible number of combinations and permutations. Additionally, most of the web application's features are independent from each other and as a consequence most of the tests perform an action and evaluate the displayed data directly afterwards.

Desktop application
The current implementation of the Desktop application does not follow the Model-View-Controller (MVC) pattern, which makes automated testing using unit tests highly complex and practically intractable. As a consequence, the whole functionality has to be tested together with the GUI. Nevertheless, the

application needs to be tested in both communication directions - to the web service/database and to the RFID-Device. Thereby, the first case is rather usual, but the second is special in some kind hence it depends at least partly on a device simulator. Such a simulator is part of the software framework provided by the Enso Detego GmbH to simulate typical use-cases and facilitate automated testing routines.

The *label printing* module is used to print new RFID-transponders. This includes printing data visually and programming the EPC onto the label. Automated tests are useful, although in most cases any kind of device simulation will be needed. Otherwise no full automation can be achieved, whenever a user-device-interaction needs to be performed. The printer device is already simulated in several ways. Two versions of which are quite interesting for the test automation.
The first one is to write the data, which are normally sent to the printer into a file instead of printing and programming the real transponder. This can be used to test whether the ”print-event” occurred and the correct data has been submitted or not. The other option is to simulate the actual RFID-printer itself. In this case, the simulator sends the same responses back to the label printing application like a real printer would do. This could be either a success message or one of several error notifications for each transponder to print.
Verifying the GUI is also useful in a small number of cases. This can, for instance, be used to verify the correct handling of the application when a print error occurred.
Other kinds of tests still have to be performed manually in the near future. As an example, the optical print of a label needs to be verified manually because although the right information is submitted, the information might be moved over the printing borders or blurred, as well as undesirable ink lines, dots and similar might be printed due to the printer’s configuration. Whereas, this step could be automated, for example with an optical verification system, this is considered out of scope within this master’s thesis.

Mobile device application
Known from previous attempts within the Enso Detego GmbH, debugging and test automation in form of unit tests is not practicable on the Windows CE 6.0 OS. Nevertheless, certain automatic system tests would be a great feature. The mobile device application can be updated automatically from a central place with a deploying mechanism over the network. As part of that, the configuration is done automatically and might run into an error for several reasons. In this case, an automatic start of the mobile device application, performing just one example task would give already a huge amount of confidence.
To perform any process correctly, the application has to work properly and the connection to the web service and the database must be established. It has to be evaluated whether any kind of functional GUI test could be applied in this case. The idea is to start the application via a remote tool while the

mobile device is connected to a PC via a cable. Although testing the application running on the mobile device, the test would be executed on a PC using a capture-reply tool in combination with a remote-control software for the mobile device.

3.5 Test Frameworks and Tools

There is a high number of possibilities for test frameworks and test tools on the market for different purposes. Each of them has more or less features and might be useful or not, depending on the use case. The selection of the appropriate test environment is important, since this determines whether the desired test cases can be implemented efficiently or not.

3.5.1 Unit Tests

As previously stated in Section 3.3.2, a number of unit tests has already been implemented for the ***web service***. These are based on the Visual Studio Unit Testing Framework. An alternative would be for instance the NUnit tests, but the testing framework of Visual Studio has been evaluated as sufficient previously and might be easier to integrate into the automated build process within the TFS, too. Hence, the TFS will be used in this thesis, as there are no reasons to change the framework from the author's point of view.
The ***web application*** is naturally separated into code executed on the server- and the client-side. There are only a few basic functionalities implemented on the server, such that the unit tests within this master's thesis will focus on the client-side. For the test of the client-side functionality, the QUnit[6] framework is used.

3.5.2 GUI Tests

For the ***web application*** these kind of tests will be based on the Telerik Testing Framework (TTF)[7]. This framework allows a code-based implementation of functional GUI tests and supports the Telerik and some of the Kendo UI features.
Unfortunately, the TTF only supports Windows Presentation Foundation (WPF) applications but not those which use WinForms, like the ***desktop application***. Such applications can be tested with Microsoft's Coded UI tests[8].

[6]Last retrieved on 30.01.2015 from http://qunitjs.com/
[7]Last retrieved on 18.04.2015 from http://www.telerik.com/teststudio/testing-framework
[8]Last retrieved on 07.11.2014 from http://blogs.telerik.com/winformsteam/posts/12-10-12/build-stronger-applications-with-coded-ui-tests-for-winforms-today.aspx

For the ***mobile device application***, the intent is to use a GUI test to ensure an established connection between the application and the web service, as the mobile device can not be addressed directly for GUI tests. The idea hence to perform remotely driven tests on the mobile device using EveryWAN as remote connection tool.

Chapter 4

Implementation

This chapter discusses the practical part of this master's thesis. The implemented tests as well as several issues are described in detail.

4.1 Collaboration Platform - TFS

As briefly stated in Chapter 1, Section 1.2.2 the collaboration platform used within this master's thesis is a TFS. This platform can either be used as service within the cloud, which is kept up to date automatically, or may be hosted on a dedicated server on premises. For the purposes of this master's thesis, a dedicated TFS within the corporate network is used as collaboration platform.

This platform needs a domain controller. To provide collisions with the in-house network, a TFS 2012 and a client have been set up as virtual machines in an isolated network. In this case, after the installation the domain controller including the users and permissions must be configured. A number of tutorials for this are available on the Internet.
The TFS has a quite complex architecture and the configuration turned out a bit challenging, as some actions are not intuitive at least for the author of this master's thesis. To clarify, the configuration is divided into three different parts. The basic TFS features can be configured within the *Administration Console*. Later on *Team Projects*, which contain the projects' source code, can be created within a Visual Studio installation connected to the TFS and only there. Such instances are typically clients, whereas the configuration of those Team Projects, including the permissions of the persons working with the code or the configuration and maintenance of so-called *Work Items* must be done via the web view of the Team Project. Work items are the fundamental elements of the TFS' ticket system. Nevertheless, adding and changing work items can be done either via the web view as well as any Visual Studio instance as described above.
The version control has to be specified when adding a new Team Project - Team Foundation Version Control (TFVC) and Git are supported. The first approach was to use Microsoft's own version control TFVC. This is a centralized version

control system similar to Apache Subversion (SVN), which had been used up to now by the Enso Detego GmbH. It turned out that the migration of the already existing code from the existing SVN repository to TFVC is an extremely tedious and error-prone process. The most effective migration seemed to be from SVN to Git and later on from Git to TFVC. Git is, as a decentralized source control system, more complex but has several advantages compared to a centralized system from the development team's view. Therefore it has been decided to use Git as version control system within the TFS. It turned out that only the cloud solution of the TFS 2012 supports Git, but not the native one. The newer version 2013 does, therefore, the server was updated. It should also be mentioned that only the Visual Studio 2013 or a newer version can connect to the TFS 2013 via the team explorer.

4.2 Web Service

Since the web service is the heart of the detego® SURVEYOR, those tests were developed first.

4.2.1 Preliminaries and Prerequisites

As described in Chapter 3, Section 3.3.2, a number of unit tests regarding the web service have been implemented in the past but not all of them still work as expected. Therefore, the initial step in order to complete them was to identify which use cases are already unit tested and to select those which do not work as expected. The existing tests already covered the most important processes but have been designed while implementing and improving the functionality of the web service. They are not suitable for automated tests because nearly all of them were positive tests and assertion statements were hardly used. Accordingly, all of the existing tests have to be revised and at least expanded with assertions to determine automatically whether the test runs pass or fail.

Another issue is the independence of the unit tests. The already existing tests had been built up depending on each other, and they needed either a certain order or a manual revision. Although unlikely for unit tests, there may be reasons for dependence, for example if the test initialization takes a long time or a huge amount of data has to be transferred. No such reason could be found for this software project. Therefore, the tests will be reworked such that they can be executed separately. This implies that a test initialize method is required, where items are transferred into certain states.

The back-end of the detego® SURVEYOR is a database and all the applications interact with this database over the web service. The instance of this database, needed for the tests, is created as an empty database in the correct form at first and to insert some predefined values, which are always the same for this software project (i.e. business steps and read points), in a second step by executing a script. It had been decided to make the tests executable independently from any additional manufacturing data stored in the database. This assures that the tests can be executed for example on a live system if any problems occur. Nevertheless, if the amount of stored data is huge, the tests may become slow. Therefore, commonly, the tests will be executed against a database created with the script mentioned before.
Some data (in the following called master data) are needed initially. In practice, these data are imported from the merchandise information system but to ensure their independence, they are created automatically before the first test is started. Additionally, with this independent master data the assert statements in the individual tests are easier to express. As an example, products and items have a 1:n relation in the system. Whenever using a product which has been created immediately, one can be sure that there are no other items stored as those created within the actual tests. Otherwise, the assertions would need to include the time span of the running tests, which makes them more complicated. These values, especially those which have to be unique in the database, are chosen unrealistic although valid, such that collisions with perhaps already stored data are unlikely.

4.2.2 Test Implementation

Unit test frameworks nowadays provide class and test initialize methods.
It was decided to use a *class initialize* method to create the used master data mentioned above, which are used for the whole test run. For the purpose of this master's thesis two different products, a supplier and some other associated data like the supplier's address and site are stored during the initialization.
Within the *test initialize* method, several items are brought into a certain state, e.g. sold or available on the stock, and so on. The service tests depend on the state of the items, therefore not each operation is valid for each state.

In the first step, the existing test methods have been evaluated regarding whether they are still applicable or not due to the change of functionality. The ones still valid have been extended with assertions. These two steps cost the most time effort for the service unit tests within this master's thesis. Assertions are done by evaluating the state of the item after the test within the database. Since the existing tests were nearly completely positive, the test suite has been expanded with negative tests for these existing methods. Finally, some missing tests have been written.

The tests are performed according to the following principle: a service method is called for a specific item, and later on the state of the item is verified. Thus, only a few of the tests are described in the following. In this section, a product is equal to an article.

public void InitializeItemValid()
The item initialization is the first operation within an item's life cycle. The product corresponding to the item's identifier (EPC) must be stored within the database. Listing 4.1 shows the test, which verifies first whether the product is stored in the database and then calls the service method with one specific identifier. Since the identifier is new to the system, it must be stored in the database as initialized item. Subsequently, a so called Track and Trace event (for logging and monitoring purposes) is stored in the database.
The tables and views of the database are used to verify the item's state. The chosen product as well as the right business step, read point and location for the initialization are verified.

```
[TestMethod]
public void InitializeItemValid()
{
  string gtin = gtin2;
  using (var entities = new SurveyorDataStoreEntities())
  {
    Assert.IsNotNull(entities.Products.FirstOrDefault(p
       => p.Gtin == gtin));
    List<string> identifiers = new List<string>();
    identifiers.Add("303487A23815B38000000001");
    using (var client = new DataServiceClient())
    {
      client.ItemInitializeEpc(

        businessStepItemIntialize,
        locationSalesFloor,
        readPointRelocation,
        DateTime.UtcNow, identifiers);
    }
    AssertItemCreation(identifiers, gtin, entities);
  }
}
private void AssertItemCreation(List<string> identifiers,
    string gtin, SurveyorDataStoreEntities entities)
{
  using (var entitiesReport = new
     SurveyorDataStoreEntitiesReport())
    {
        foreach (var identifier in identifiers)
        {
            var itemEntity = entities.Items.FirstOrDefault(i =>
                i.Identifier == identifier);
            var product = entities.Products.FirstOrDefault(p =>
                p.Gtin == gtin);
            var stockByItem = entitiesReport.StockByItems.
                FirstOrDefault(st => st.Identifier == identifier
                && st.Culture == "de-DE");
            var trackAndTrace = entitiesReport.
                TrackAndTraceBriefs.FirstOrDefault(tt => tt.
                Identifier == identifier && tt.Culture == "de-DE
                ");
            var movemByItemAndBS = entitiesReport.
                MovementByItemAndBusinessSteps.FirstOrDefault(
                mib => mib.ItemId == itemEntity.ItemId && mib.
                Culture == "de-DE");

            Assert.AreEqual(product.ProductId, itemEntity.
                ProductId);
            Assert.AreEqual(businessStepItemIntialize,
                stockByItem.BusinessStepName);
            Assert.AreEqual(businessStepItemIntialize,
                trackAndTrace.BusinessStepName);
            Assert.AreEqual(locationSalesFloor, stockByItem.
                LocationName);
            Assert.AreEqual(readPointRelocation, stockByItem.
                ReadPointName);
            Assert.IsNull(itemEntity.Expired);
            Assert.AreEqual(businessStepItemIntialize,
                movemByItemAndBS.BusinessStepName);
        }
    }
}
```

Listing 4.1: Valid initialization of an item

As a negative test, the same method is called with an identifier of an unknown product, which just has to be ignored. After the test the identifier is not allowed to exist within the database. Another negative test tries to initialize an existing item currently being in another business step. Again, this initialization has to be ignored. This is verified, such that the business step has not been updated by the test.

These kind of tests have been implemented for the most critical components of the SURVEYOR platform to ensure that the basic functionality can be tested in an automated way. Implementing tests for the whole service would be out of scope of this master's thesis.

public void DoubleBulkInsert()
This test is used to simulate two stock takings performed simultaneously. Due to the architecture of detego® SURVEYOR it is possible that the stock taking process is completed on two mobile devices at the same time. It has to be assured, that the data of both processes are handled correctly. Thus, this multi-threaded test calls the same method with the same 1000 items. Within the database two distinct groups with the same items must be stored. Listing 4.2 shows the according code.

```
[TestMethod]
public void DoubleBulkInsert()
{
  DataServiceClient client = new DataServiceClient("
     BasicHttpBinding_IDataService");
  var t1 = System.Threading.Tasks.Task.Run(() =>
     InsertStockTaking(client));
  var t2 = System.Threading.Tasks.Task.Run(() =>
     InsertStockTaking(client));
  System.Threading.Tasks.Task[] tasks = {t1, t2};
  System.Threading.Tasks.Task.WaitAll(tasks);
  Thread.Sleep(3000);

  using (var entities = new SurveyorDataStoreEntities())
  {
    DateTime now = DateTime.UtcNow.AddSeconds(-3);
    var groups = entities.Groups.Where(g => g.GroupType
       == 3 && g.TimeStampCreated >= now);

    Assert.AreEqual(2, groups.Count());
    Guid Id1 = groups.FirstOrDefault().GroupId;
    var groupItems = entities.GroupItems.OfType<
       GroupItemIdentifier>().Where
      (gii => gii.GroupId == Id1);

    Assert.AreEqual(1000, groupItems.Count());
    Guid Id2 = groups.OrderByDescending(g => g.GroupId).
       FirstOrDefault().GroupId;
    groupItems = entities.GroupItems.OfType<
       GroupItemIdentifier>().Where
      (gii => gii.GroupId == Id2);

    Assert.AreEqual(1000, groupItems.Count());
    Assert.AreNotEqual(Id1, Id2);
  }
}

private bool InsertStockTaking(DataServiceClient client)
{
  List<string> identifiers = new List<string>();
  for (int i = 8000; i < 9000; i++)
  {
    identifiers.Add("303443D11C0AD9C06B" + i.ToString("D2
       ").PadLeft(6, '0'));
  }
  client.InsertStocktaking(businessStepItemInventory,
     locationBackroom, readPointHandheld, GroupType.List,
      DateTime.UtcNow, null, identifiers);
  return true;
}
```

Listing 4.2: Simultaneous performed stock taking with 1000 items

As previously described, each business step has in and out states. These are also mapped within a database view. To verify that these conditions are still fulfilled, the business steps are performed on items in specific states.

To gain some confidence about the data delivered by the database views and to ensure that these are still working after a modification, one integration test was developed in addition to the unit tests described above. As it is performed within the service, it is mentioned here. Initially, an ordered test of the service tests mentioned above is executed, and later on the test verifies whether the data in the database views are still correct. For this, these views are used which display the actual amount of items within the stock. For the verification values from other database views, providing the same information, are used.

4.2.3 Problems

The only problem occurred at the start of the multi-threaded test development. At first, it has been tried to start as 4.3 shows, but the threads did not run simultaneously.

4.2.4 Results

The implemented unit- and integration tests cover the critical functionality within the detego® SURVEYOR platform and build the foundation for automated regression testing. In particular, the tests provide a code coverage of about 40%. This is due to the fact that this master's thesis does not cover all processes of detego® SURVEYOR and because there is still death code available. The code coverage needs to be extended in the course of further developments. Additionally, the tests concerning the database views might be revised in order to get more but smaller tests and to expand them on the other database views in the near future. Nevertheless, the approach and implementation described above increases the efficiency and provides a considerable speed-up compared to the previously used, manual testing procedures.

```
public void DoubleBulkInsert()
{
  Thread thread1 = new Thread(new ThreadStart(
     InsertStockTaking));
  Thread thread2 = new Thread(new ThreadStart(
     InsertStockTaking));
  thread2.Start();
  thread1.Start();
}
```

Listing 4.3: Erroneous try of a multi-threaded test

4.3 Web Application

The web application is tested with a unit testing framework for JS and functional GUI tests.
The unit tests are used to test the functionality implemented on the client-side with JS code, for example whether the request's response provides the correct data. The aim of the functional GUI tests is to verify the displayed data after any user interaction and that no unexpected behavior occurs after a user's action. Some use cases might be treated with both kinds of tests. Since unit tests can be performed faster than any GUI tests, they should be preferred.

The web application communicates via the web service with the database. Thus, the web application tests can only be executed if the web service and database are installed and configured accordingly and the connection is established.

Again, initial data are necessary for both kinds of tests. The main focus for testing the web application lies on the various reports that require initial data in order to be tested in a meaningful way. There are two other possibilities for an initial data set-up. The first approach is to use a database which has been populated by a series of preceding actions (method calls). In contrast, the second approach involves the backup of a productive database. In the first case, the service tests would not be allowed to clean-up the database after their execution, which endangers the independence of the web service unit tests and makes them more complicated. At the same time, these web service tests do not insert that amount of data which might be useful or even necessary for some of the web application tests. Therefore, it was decided to use a specific backup of an already productive system of this project.

4.3.1 Unit Tests

QUnit is the used framework for this kind of tests. Since the tests use the web service and database as backend instead of a mock-up, the definition of unit test must be reconsidered. Nevertheless, one test always deals with one unit or, respectively, one function of the JS code. It would be possible to provide such a mock-up as backend, but this is beyond the scope of this thesis.

4.3.1.1 Preliminaries and Prerequisites

When starting to implement the first QUnit test, the question how to integrate the tests into the web application the best way was faced. Developing the test script within the web application itself, as it is done within several tutorials, was no possibility. The tests would be executed at the customer's installation too, which is not desired. Taking a copy of the web application for the test implementation would implicate high maintenance costs and is error-prone, because with each modification the copy has to be updated too. Finally, it was decided that the best solution would be to load the content of the web page dynamically into the tests' Hypertext Markup Language (HTML) file when starting them.

Within this subsection the terms *web page* and *page* refer to one page within the web application, i.e. the report *Stock by Articles* which lists all available items grouped by the article. The functionality of the reports is mainly the same. Certain data are displayed and may be filtered or exported. It should be able to adapt the tests of one report easily to the others. The functionality of the reports is divided into different JS files. There is one general script file (survejor.js) providing basic functionalities used within the whole web application, and each web page has an own JS-file too. Within the scope of this master's thesis two kinds of tests have been developed. On the one side functionality within the general script file has been tested with several input possibilities for a function. On the other side, the assertions of the individual page tests were done with hard-coded values as it was defined to set-up the database with the same backup every time. Nevertheless, these assertions could be extended to query the database for the right value. This would cost additional time and since the data for the assertions are known, it was decided to use those.

4.3.1.2 Test Implementation

The most time for the test implementation has been investigated at the beginning until the content of a report could be loaded dynamically into the test's HTML-file. A minimalistic QUnit example is shown in Listing 4.4. To test this web application, the content of the script surveyor.js can be added by including the script into the HTML's body. To get access to the response of any Asynchronous JavaScript and XML (AJAX) request and therefore to the retrieved data, the content of the page's JS-file has to be appended to the HTML-tag <div id="qunit-fixture">. This lasted several days, although the resulting code is simple. The occurring problems are elaborated in more detail within Section 4.3.1.3.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Tests</title>
  <link rel="stylesheet" href="//code.jquery.com/qunit/
     qunit-1.16.0.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="//code.jquery.com/qunit/qunit-1.16.0.js"><
     /script>
  <script>
  QUnit.test( "my first test", function( assert ) {
    assert.ok( 1 == "1", "Passed :)" );
  });
  </script>
</body>
</html>
```

Listing 4.4: A basic example for a QUnit test

QUnit provides a ”beforeEach” and ”afterEach” function, which are executed before and after each test. These were not necessary for this test purposes, but to assure that all scripts are loaded completely before starting the first test and the session was renewed by passing an authentication process, the tests were configured not to start automatically.

The web page’s script to test has to be adapted in a slightly way, such that the data source is no longer accessible just for internal methods. The data source fills the data grid of each report and is probably needed in each test. Thus, the viewModel of Kendo must be expanded. How to access the data source is shown in line seven of Listing 4.5.

QUnit.test(”load data source”, function (assert){...}
The basic functionality of a report is verified within this test. When calling the page, the data source to display within the grid is requested from the database. This test verifies whether the first grid page (per default containing 20 columns) is filled with data. Since, JS acts asynchronously, the test must be stopped until the data source has been loaded. Otherwise the assertion might be done before the request to the database is completed. Listing 4.5 shows the test with two assertions.

```
 1 QUnit.test(”load datasource”, function (assert) {
 2   expect(2);
 3   var done = assert.async();
 4   var done1 = assert.async();
 5 
 6   setTimeout(function () {
 7     assert.notEqual(ds.page.viewModel.productsSource,
         undefined, ”viewModel loaded correctly”);
 8     done1();
 9   }, 10000);
10 
11   ds.page.viewModel.productsSource.bind(”change”,
       function (e) {
12     assert.equal(e.items.length, 20, ”20 items loaded
         into datasource”);
13     done();
14   });
15   ds.page.viewModel.productsSource.read();
16 });
```
Listing 4.5: QUnit example stopping the test while loading data

QUnit.test(”download PDF”, function (assert){...}
The PDF download queries the data source for the report from the database

and exports it to a PDF-file. This functionality is verified in two steps. First, the test has to wait until the data grid has been filled with the data. Usually the PDF is exported when the user clicks on a specific button but the test performs the AJAX-call directly. It verifies if the response of the request is valid. To check the amount of data items in the response, the URL is modified, such that the response is of the type JSON and the amount can be retrieved.

QUnit.test("download CSV", function (assert){*...*}
This test does nearly the same as the one above but exports the data source as CSV-file.

QUnit.test("Multiselect supplier", function (assert){*...*}
This test has been implemented as example for testing the multi-select filters. As for the tests above, the data source must be loaded first, later on a supplier is selected and the data source is filtered such that only articles from this supplier are shown. The test verifies the name in the supplier's column of each row in the data source with the selected name.
Executing this test together with the other tests sometimes leads to a timing problem. The assertion starts before the data source is loaded completely. What the problem is could not be figured out until now.

The following tests describe how to verify general functionalities within the surveyor.js and without loading a specific page. Thus, the code within Listing 4.6 (the dynamic page load) is not necessary here. The QUnit tests start right after the authentication process. These tests aim to verify the words/sentences displayed on the page.
To support different languages the detego® SURVEYOR uses a lookup mechanism. Within the source code is the key, which is replaced with the appropriate language term at run time (localization). There is also a fallback, if the key does not exist in the database. If the given key can not be found, the fallback splits the given key at the first point and searches for the second part of the key in the lookup table. This is repeated until a value is found or the string is empty.
Another feature is the tooltip displaying the whole term for column headers, whenever the values' abbreviation is used.

QUnit.test("Test localizing fallback - Displayname found", function (assert){*...*}
The test selects the first string including *.Column.* within the key from the database and adds a prefix, which definitely does not exist. With the resulting string the localise-function is called. It must return the according value of the key selected before.

QUnit.test("Test localizing - Displayname found", function (assert) {*...*}
This test does the same without prefix. This is the default case and should never fail, otherwise it must be suggested that the whole web page is displaying

the keys instead of the appropriate localization.

QUnit.test("Test tooltip of short header - Abbreviation != null", function (assert) {...}
As the tests above the database is queried for a string. The tooltip function is (at the moment) only used for the columns headers, thus the used key has to include the term *.Column.* again, and additionally, the abbreviation must not be *NULL*. The function - gridHeaderTemplate(...) - which returns the appropriate HTML-tag is called and the result is compared with a partly hardcoded HTML-tag which uses the same key and abbreviation.

The HTML-tag for the grid's column headers depends on the parameters of the gridHeaderTemplate-function. The icon-class and the key are obligatory and the third optional parameter defines the header's text if a tooltip should be shown. Without optional parameter the header has an icon and the value, otherwise a single icon or an icon and the abbreviation is shown. These cases are each tested within an own test, where both cases, whether the abbreviation in the database is NULL or not, are considered. Additionally one test exists handing over an invalid string as optional parameter, which has to results in an icon and the value.

4.3.1.3 Problems

The main problem at the beginning was, that the page content could not be loaded dynamically, and it was difficult to figure out, what the problem was. At the beginning it has been supposed, that the problem occurs in combination with the KendoUI framework of Telerik, which turned out unfoundedly. The finally and working function which append the content is provided in Listing 4.6. The first problem, which still has not been solved, was the identification of the html <div id="content"> of the web page to append. The workaround for this can be seen in line six of Listing 4.6. Later on, to get the script an AJAX-call as in Listing 4.7 has been used and failed.

After these problems have been solved, the test started before the content had been loaded completely. Fortunately, QUnit provides a configuration which causes the tests not to start immediately when the test-HTML is called. Initially the tests' start has been called to early, which caused a race-condition sometimes.

Additionally, the web application takes over the browser's default culture at the beginning. For some reason this failed within the tests. This issue could be easily fixed by setting the culture to english within the surveyor.js, whenever the culture-object is null at the page load.

```
function loadStockByProducts(){
$.ajax({
  url: '../../../Reports/stockByProducts',
  dataType: 'html',
  success: function (response) {
    var div = $(response)[75];
    $('#qunit-fixture').append(div);
    $.getScript('../reports/stockByProductsViewModel.js')
    .done(function (script, textStatus) {
      console.log(textStatus); // Success
      console.log("ViewModel loaded");
      QUnit.start(); //Call here to provide a race
          condition
    })
    .fail(function (jqxhr, settings, exception) {
      console.log(exception);
      console.log(jqxhr.status);
      console.log("failure");
    });
  }
});}
```

Listing 4.6: Function loading the page's content dynamically

```
$.ajax({
url: '../reports/stockByProductsViewModel.js',
dataType: 'script',
success: function (e) {
  console.log(ds.page.viewModel);
  kendo.bind($('#content'), ds.page.viewModel);
}});
```

Listing 4.7: Failing AJAX-call to load the web page's script

Another issue has been faced while the test's implementation, caused by a wrong authentication process within these. This resulted in an expired session after some time of testing and the browser's console showed the failure "Uncaught TypeError: Cannot read property 'dataSource' of undefined@ 1351 ms". Hence the failure occurred within the source code of a library, it took some time to figure out what the problem was.

4.3.1.4 Results

Once the problems at the beginning have been solved, the tests could be implemented without heavily problems. One aspect which must be kept in mind is the asynchrony nature of JS and thus also QUnit. The tests must be stopped at certain steps, to wait for a result. QUnit is going to deprecate several core methods and replaces them, within the release 2.x. Nevertheless, the new methods can be used since QUnit 1.16.x, which has been used for this master's thesis. One of these methods is *QUnit.asyncTest()*. Instead *var done = assert.async()* is used within a synchronously test, which waits until *done()* is called. Most of the cases a test fails while the development it is caused by the asynchronism.

4.3.2 Functional GUI Tests

For the GUI tests the TTF is used. These tests are implemented to verify that specific user interactions do what they are supposed to do, i.e. the web browser displays the correct data after any actions have been performed by a test.

4.3.2.1 Preliminaries and Prerequisites

For the first tests one of the simpler reports was chosen. Since the TTF was new to the author of this master's thesis, the initial training should base on some simple actions. Furthermore, it was not certain at the beginning whether the implementation of the tests would be possible with this framework, although it seemed to be the best alternative. Nevertheless, the idea was to adapt these first tests to more complex reports afterwards. The reports which should be tested have some features in common. Each provides the data in a grid with up to three levels and the possibility to hide the lower levels. Some of the grid's columns are hidden by default and the user can show and hide all individual columns via a drop-down menu. Additionally every report provides two ways of filter options. The first one are multi-selects or time-picker for some predefined columns. These are easier to apply for the user as the second method, which is the default filter option within the grid and available for nearly all columns. The last consistent feature is a export function of the displayed data as PDF- or CSV-file.

In general, the tests should verify that the web page behaves accordingly after any user actions, such that the file exports, show/hide columns and every filter are working correctly. The latter one means that the data changes according to the filtered values. It is not the aim of these tests to verify every possible combination of the filters. Hence this can be done much easier and faster within QUnit tests. For the multi-select filters it is sufficient to test with two different values, because there should be no reason why the multi-select works for two selections but not for any more.
The columns' filters are constructed differently depending on the data-type

of the value. For example a string value can be filtered whether it ”contains”, ”starts with”, ”ends with”, ”is equal to” or ”is not equal to” any given string. The filters are performed via typical select queries in the database and therefore it is not necessary to test every possibility. It is assumed that the SQL functionalities work by default and consequently, that if one selection works as required, the others do as well. The typical problem which occurred in the context of these filters until now has been that a table or view within the database changed and does not correspond to the web application anymore. Whenever this happens, at least one filter does not work at all.
The file export tests have the purpose to verify the download of files (PDF or CSV). The content of the downloaded files will not be parsed and evaluated as it is completely out of scope for this master’s thesis.

The TTF supports four web browsers, namely the **Internet Explorer**, **Mozilla Firefox**, **Safari** and **Chrome**. Considering the every-increasing market share of Google’s Chrome browser, the tests are carried out mainly using Chrome throughout this thesis. Nevertheless, changing the web browser for the tests can be performed easily (changing one line of code within the test initialization) and the tests should work in the other browser too. Since the distinct web browsers sometimes work differently, some other adaptations might be necessary that the individual tests work correctly.
Additionally, it is recommended to adapt the web browser’s settings on the testing machine as described in the documentation of the TTF[1]. Especially the provided download functions, which are going to be used within this master’s thesis, behave differently depending on the settings of the used browser. For Chrome it is recommended to set the download option to ”Ask where to save each file before downloading”.

4.3.2.2 Test Implementation

The TTF provides a basis test class which facilitates the set up and tear down methods and provides some other functionalities for example getting the active browser. To run the unit test using the TTF, an initialization code is needed. This initialization is executed before each test.
At first, the Initialize()-method from the base class is called. This method has several overloads, but for this master’s thesis the one in Listing 4.8 is used. The parameters define whether the browser is going to be reused, the log location and a write line delegate to call when logging from Log.WriteLine. The first parameter is set to *true* which means that the opened web browser for the first test is reused for all other tests within the test-run. This behavior saves time for each executed test which otherwise would be consumed by opening, navigating

[1]Last retrieved on 09.05.2015 from http://docs.telerik.com/teststudio/testing-framework/getting-started

through and closing the web browser. This feature will be used because GUI tests need quite some time anyway while performing AJAX calls, displaying the data and so on. As a consequence, the login has to be performed after launching the browser before the first test run is started. The web browser will be closed after all tests has been executed.
Next, the function SetTestMethod() is called. Together with the Initialize Method, this procedure is responsible for the WebAii initialization which is used by the TTF.
Finally some additional initialization actions are performed. First, a new Chrome browser is launched. This line of code will be ignored if a browser has been already launched within the test run, caused by the Initialize()-method described above. Afterwards, the options to automatically refresh the DOM-Tree and to wait until the browser is ready are set to true. Finally the web browser navigates to the report to test (Listing 4.8). This last statement causes, for all except the first test, a page reload, which is necessary to undo the actions performed in the test before.

```
Initialize(true, "C:/WebAiiLog", new TestContextWriteLine
    (this.TestContext.WriteLine));

SetTestMethod(this, (string)TestContext.Properties["
    TestName"]);

Manager.LaunchNewBrowser(BrowserType.Chrome);
ActiveBrowser.AutoDomRefresh = true;
ActiveBrowser.AutoWaitUntilReady = true;
ActiveBrowser.NavigateTo(TESTPAGE);
```

Listing 4.8: Test Initialization within the TestInitialize()-method

```
KendoGrid productsGrid = Find.ById<KendoGrid>("
    productsGrid");
Element headers = productsGrid.Find.ByNodeIndexPath("
    1/0/0/1/0");
IReadOnlyCollection<HtmlAnchor> pdfUrls = Find.
    AllByAttributes<HtmlAnchor>("data-bind=attr: { href:
    exportUrlPdf }");
HtmlButton bttn = filter.Find.ByExpression<HtmlButton>("
    type=submit", "class=k-button k-primary");
```

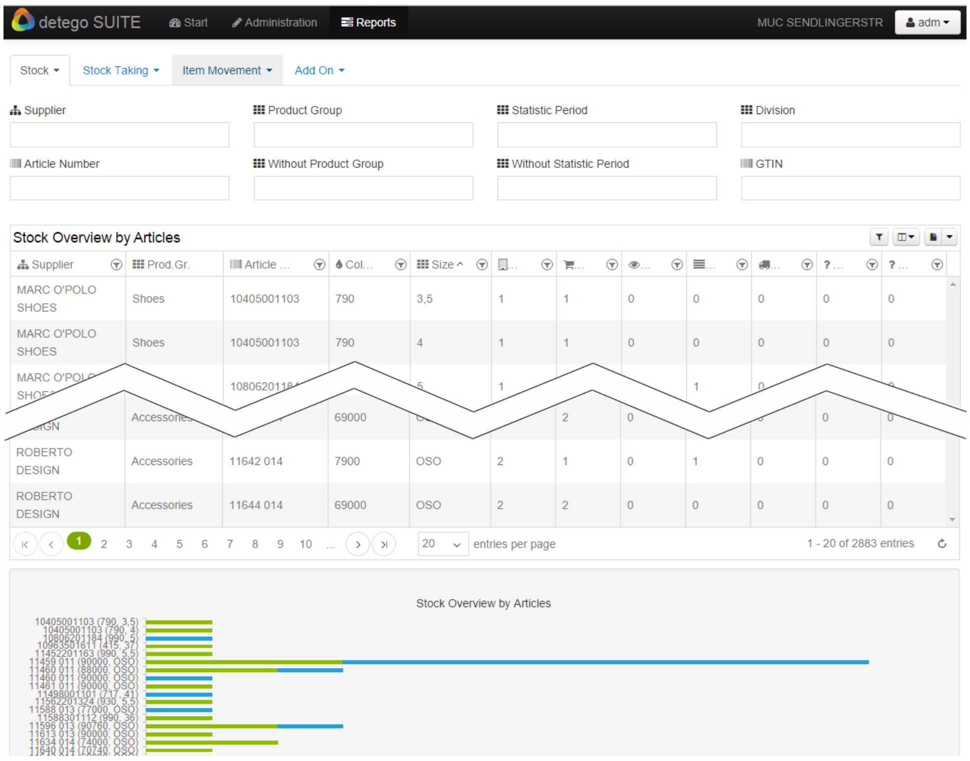
Listing 4.9: Some of TTF's methods to find an object

The inevitable feature for GUI tests is to identify any element within the Document Object Model (DOM) tree. Therefore, the TTF provides a number of methods to identify any HTML element on the displayed page in the web browser. These methods can either be applied on the whole page or any already identified element, for example a <div>. Listing 4.9 shows some of the identification possibilities, either returning exactly one or all elements matching the criteria. These are handed over as parameters.

The implemented tests are described below and base on some reports. In the following, the terms *article* and *product* are used synonymously.

Report: Stock Overview by Articles

The first report, *Stock Overview by Articles*, is shown in Figure 4.1 and lists all articles in a table. Additionally, a bar chart represents the amount of articles, as a bar for every row in the grid.



Supplier	Product Group	Statistic Period	Division
Article Number	Without Product Group	Without Statistic Period	GTIN

Stock Overview by Articles

Supplier	Prod.Gr.	Article ...	Col...	Size ^	...	...	...	...	...	? ...	? ...
MARC O'POLO SHOES	Shoes	10405001103	790	3,5	1	1	0	0	0	0	0
MARC O'POLO SHOES	Shoes	10405001103	790	4	1	1	0	0	0	0	0
MARC O'POL SHOE		1080620118		5	1			1	0		0
IGN	Accessorie		69000			2	0			0	0
ROBERTO DESIGN	Accessories	11642 014	7900	OSO	2	1	0	1	0	0	0
ROBERTO DESIGN	Accessories	11644 014	69000	OSO	2	2	0	0	0	0	0

1 2 3 4 5 6 7 8 9 10 ... 20 entries per page 1 - 20 of 2883 entries

Figure 4.1: Screenshot of the report "Stock Overview by Articles"

StockByProdBasisTest()

This test calls the stock overview by articles' report and identifies the data grid in it. Based on that, it evaluates whether the initial data are displayed correctly and some basic features of the report.

At first, the data grid is identified and the displayed data of the grid are evaluated. This evaluation focuses on the data representation. To clarify, some properties are mapped as integers in the database, but must be displayed as humanized string. As an example, the *ProductGroup.100* should be displayed as *Heavy knit*. There are five such mappings in the database. The test iterates through every cell of the grid, regardless whether the column is hidden or not. The reports have a paging system where, per default, 20 rows are displayed on each page. While iterating through the rows, the bar chart is inspected too. Every displayed row has to be represented there. The uniqueness of a row is given visibly through the article number, the color number and the size, which are also the labels for each bar in the chart.

Next, the test iterates through all available columns and verifies that some specific ones have a filter option. The columns' representation in the database is defined hard-coded in an array. First, it was considered to access and reuse Kendo's data source, where the columns of the report are defined. Unfortunately, the TTF does not provide this feature at the moment, or at least it could not be figured out how to access these data. This implies, whenever the columns of the data grid change, for example after implementing a new feature, the test must be updated too. However, the test does not depend on the displayed names of the columns which may change several times within a project's lifetime. Columns, which have a filter option are identified over the class attribute ”k-filterable” in the column header. In return, all the other columns are not allowed to have this attribute. Those, for which the filter option is available, are predefined and therefore also hard-coded.

The third important functionality in this context is the PDF and CSV export of the data. For this, the name of the file and the path to save are defined within the test. Unfortunately, the download function of the TTF depends on the browsers settings. As described above, it is recommended to change the settings such that the browser asks where to save the file. In a first try, this has not be done. The download has been performed accordingly, but the given file path has been ignored completely and the default path defined within the web browsers settings has been used. This makes it more complicated to verify whether the file has been downloaded or not. Whenever this path is not the same as used in the code, the code has to be adapted, which obviously is not required for automatic tests. Therefore, the browsers settings on the testing machine used for this test have to be changed. The test creates the given path if it does not exist and deletes the file with the given name if it exists. It verifies again that the file does not exist before the download is performed and verifies the contrary afterwards.

All the above described functionalities for this test method belong to positive testing. Hence, they test the basic functionality of this report and do not depend on user input, negative tests would not provide any additional information.

StockColumnsTest()
This test verifies whether every column, which is hidden by default, can be displayed. All columns are shown in a drop-down menu and the user can hide/show the column by clicking on the according column name.
The test iterates through all columns, recognizes, which is not shown and clicks on it. After that, it verifies that for each row in the grid all columns are set to

be visible. At the end, the reverse test is executed, such that all columns are set to be hidden to the user and the verification of that is done too. Again, these are positive tests.

MultiSelectArticleNumberTest()
This test filters the grid data using the multi-select option of article numbers. A drop-down offers all article numbers for user selection.
The test selects two article numbers (one after each other as it would be done manually) and verifies the data in the grid afterwards. The grid must not show any rows with other article numbers. At this point, it is not necessary to repeat, for example, the verification of the humanized string format, which is tested in the StockByProdBasisTest(), because the filtered data are just a subset of data. The verification, whether the amount of displayed data is correct, can be done much easier within a QUnit test.
In some rare cases the selection has not been done. The reason for this has not been found. As a workaround, in case of a failure, the test must be done manually again to verify whether there is actually a bug in the application or not. Nevertheless, whenever the test passes, the developer can be sure that the application is working correctly within this aspect.

Duplications: Since multi-selects are available in most reports, the duplication of this test would be useful and necessary. This report has, for example, eight multi-selects. Two of them are special cases; hence, the selection is excluded from the displayed data. It is easy to reproduce and adapt the tests accordingly.

MultiSelectWithoutProdGroupTest() is an adaptation which provides all articles without two specified product groups (Accessories and Dress).

StockColumnsFilterTest()
This test filters the grid data using the filter option within the column's header. For some columns this option is disabled by design, the others provide a filter depending on the data's type in the database. The filter value can either be a string, a numeric value, or a predefined value picked from a list.
The test has been implemented for all columns with filter options. Unfortunately, side-effects occurred while implementing and testing the test. The key value sent to the input form has not been recognized correctly in approximately every fifth test run. The reason for this failure has not been found. It is suggested, that there is a bug in the TTF, and the test might be used in the future. At the moment this makes the tests useless, because it reports a failure although there is none. Thus, it has just be implemented with one test value for every column. Once a solution for the problem described above has been found, an implementation of negative tests would be useful and necessary. Nevertheless, this would only mean an adoption of the forms' input and therefore not too much work.

4.3.2.3 Problems

The TTF seemed to be a powerful tool at first glance but the provided documentation and examples cover mainly some of the basic features. A forum is available, however, more complex actions must be given a trial. Existing bugs within the TTF are mostly recorded in the TTF's forum, including a described workaround as temporary solution.
Additionally, Telerik's Kendo UI generates a complex code structure. For example, the available options for a multi-select filter exist twice within the generated code, both as option-tags within a select statement and as unordered list-items. Another problem concerns the load of the web page and have occurred especially at the beginning of the test implementation. Whenever the test was not explicitly halted until the load progress has been terminated, the test was continuing immediately. This resulted in (partly just occasional) failed tests because any action or assertion was performed too early. In this connection it has been recognized that the TTF's command to wait for AJAX calls is ignored, at least within this web application. As workaround, the tests have been designed to wait either until the page is ready or a specific element within the page is loaded.

As mentioned in the section above, the input for a textbox is not always recognized correctly but no solution was found for this problem. Additionally, it has been stated within the description of the StockByProdBasisTest that the Kendo data source cannot be reused and must be defined hard-coded.

The test execution engine of Visual Studio has stopped working at times during the execution of the tests. This was mostly caused by a missing wait statement, too. An unsolved problem was the occasionally changing focus. When working with two monitors, it occurred that the Visual Studio was opened in one screen and the started test opened in the other. Instead of writing a string or number to the opened test browser, the test wrote the value into the opened file within Visual Studio.

It is possible that a project specific problem might occur in the future because the data for product groups and other predefined values are imported from the merchandise information system. As these imported names are displayed in the web application and must not be unique, there might also be equal selection options within the filter tests. The test assertions might get more complex for such values.

4.3.2.4 Results

Although the TTF supports a lot of features and especially those of Telerik's Kendo UI, in-depth usage revealed unfixed bugs and consequently, it is practically unusable except for simple operations. All these circumstances and known issues mentioned above imply that writing the tests may take more time than assumed. It must be estimated for every test whether the test automation brings a benefit compared to manual tests or not. Because of this, only a few tests have been implemented in this section and all of them are positive tests.

Nevertheless, during the implementation of the StockColumnsFilterTest() a bug within the detego® SURVEYOR system had been found. Actually, the bug occurred when importing the data from the merchandise information system. Some of them have not been processed correctly. As consequence, the filter option for one column did not provide any of the predefined values for selecting.

To conclude, the TTF would be a rich tool to test the Web Application with functional GUI tests if some of the issues concerning the indefinable behavior would be solved.

4.4 Desktop Application

Functional GUI tests have been used to test this application. The TTF was not applicable for these tests because it does not support Windows Forms.
Four different frameworks were tried out, whereby two (White[2] and Test Automation FX[3]) manifested themselves from the beginning as useless for this desktop application. At least some of the essential Telerik controls have not been found when executing a simple test with those frameworks.
Thus, Microsoft's Coded UI Tests framework[4] was used. This framework also does not support all, but at least the necessary Telerik controls and actions[5].
Nevertheless, after implementing some tests, it turned out that Coded UI was also not usable for all kinds of tests within this desktop application.
At last, the Ranorex[6] test framework has been used to implement the complete tests suite.

[2]Last retrieved on 07.05.2015 from http://teststack.azurewebsites.net/white/index.html
[3]Last retrieved on 07.05.2015 from http://www.testautomationfx.com/
[4]Last retrieved on 07.05.2015 from http://msdn.microsoft.com/en-us/library/dd286726.aspx
[5]Last retrieved on 07.05.2015 from http://www.telerik.com/help/winforms/codedui-supported-controls-and-actions.html
[6]Last retrieved on 07.05.2015 from http://www.ranorex.com/

4.4.1 Preliminaries and prerequisites

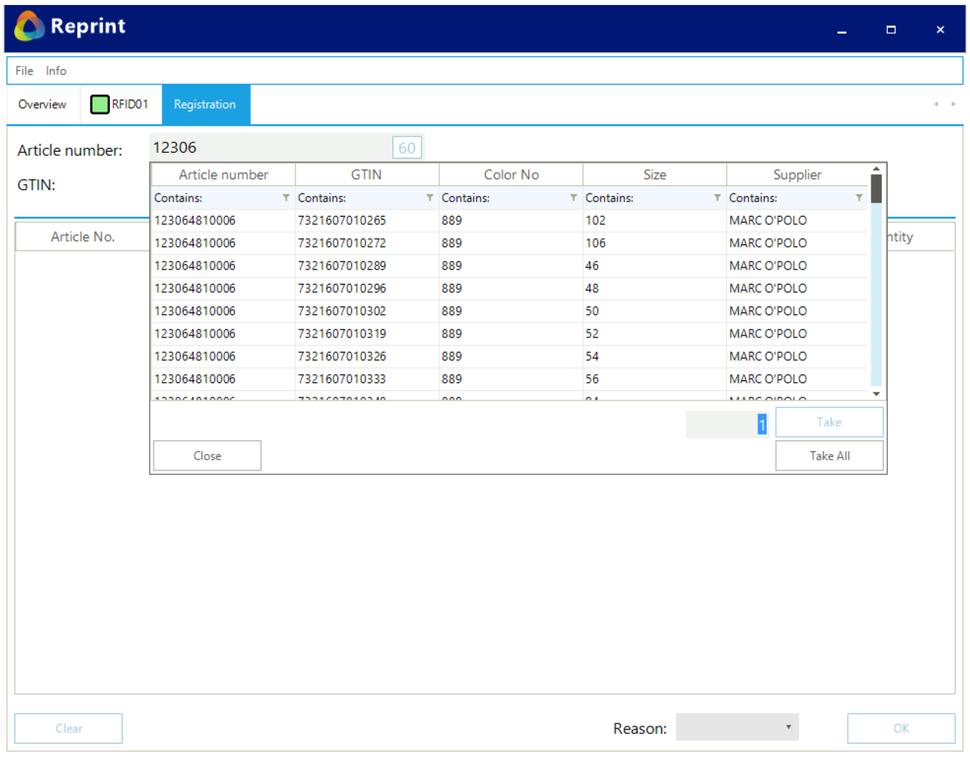
As stated above, the desktop application communicates with the database via the web service. Consequently, the database and the web service must be installed and configured accordingly to run the tests successfully. The connection between the desktop application and the backend as well as the RFID-printer must be established and the database has to be initialized with the same backup as the web application tests.

The desktop application is used to order new print jobs on one hand and on the other to print the RFID-transponders of these. However, new print jobs also can be ordered outside of the application too, for example with the mobile device, the *Registration* area of the desktop application provides the same feature. The user selects the article to print based on the article number or the Global Trade Item Number (GTIN). Within an *Overview* area, the application provides a list of all configured printers including their state (offline, ready, busy, or error) and a list of new print jobs. Additionally, for each printer, a dedicated area is available, where the print progress is displayed. Whenever an error occurs it is shown here too. Within this master's thesis one printer is configured, namely *RFIDZ01*. After a successful printing process, the transponders are written to the database and the processed print jobs are marked as completed there.

In order to work appropriately, some configurations in a XML-file must be adapted, for example the host, where the web service is running, as well as the IP-address of the printer and so on. The configurations to write within a TXT-file instead of printing onto a label or to use the printer simulator are set within this file too.

The label printing module can be categorized into the three parts *Overview*, *Printer* and *Registration*, which already have been mentioned above.
The first one presented here deals with the print job registration. There, the article to print can be selected either by inserting the article number at least partly and picking the desired article from a list or by inserting the exact GTIN. Using the article number option, the user has to reduce the amount of corresponding articles to 100 or less in order to open the list of available articles. This can be achieved by inserting the first digits of the article number, as shown in Figure 4.2. The reason for this is that more than 900 000 articles are saved within the database and selecting one out of all these is not realistic.
The GTIN can be inserted with two ways, either manually or by means of a barcode scanner. Depending on this, the application behaves differently by design. Technically, both numbers are inserted over the keyboard buffer. The application differs between those two types by evaluating the amount of the inserted digits and the time interval between two pressed keys. If the amount of digits is eight or more and the interval is below 100 ms, the application assumes

that the barcode scanner was used and inserts the article directly to the list, if the GTIN is valid. Whenever the GTIN is inserted manually, the user has to click on an Add-button additionally.
The amount of labels to print (the default is one) can be changed before adding the article to the list or afterwards within the list. To send the article list as new print job to the database, a reason why the label(s) should be printed, must be selected too.



Article number	GTIN	Color No	Size	Supplier
Contains:	Contains:	Contains:	Contains:	Contains:
123064810006	7321607010265	889	102	MARC O'POLO
123064810006	7321607010272	889	106	MARC O'POLO
123064810006	7321607010289	889	46	MARC O'POLO
123064810006	7321607010296	889	48	MARC O'POLO
123064810006	7321607010302	889	50	MARC O'POLO
123064810006	7321607010319	889	52	MARC O'POLO
123064810006	7321607010326	889	54	MARC O'POLO
123064810006	7321607010333	889	56	MARC O'POLO

Figure 4.2: Screenshot of the label printing registration tab

4.4.2 Test Implementation

Again, the class initialization code is used to start the desktop application. It is assumed that the application is installed with the set-up at the default location. Nevertheless, the path to the executable is defined as a static variable within the test source code and may be changed to any other location easily. The tests are integration tests but still independent from each other.

In order to test print errors, the printer simulator - a tool from Enso Detego GmbH - has to run with the error probability set to zero percent.

4.4.2.1 Coded UI Framework

To generate a Coded UI test, the sequence to test must be recorded with the test framework's Test Builder. This tool allows to record sequences and later on C# code may be generated from those. Additionally, the tool provides the functionality to evaluate any GUI element within the application.
In general, every recorded action or evaluated element corresponds to one line of generated code. Nevertheless, in some cases, several simple actions are combined. One example would be clicking into a textbox and inserting a number results in one code line. Listing 4.10 shows a few lines of the generated code for one test case.

The generated code may be expanded by hand. Therefore, it is possible to use the TestInitialize() section to start the desktop application before each test from a specific location directly. This saves time while the usual test's execution, in contrary, starts the application over a recorded action, navigating to the executable.

For the generated assert statements any GUI element of the application may be selected. One or more properties must be chosen and the required results have to be defined within the GUI of the Test Builder. Then the test evaluates exactly these properties of a specific element, for example whether the cell in the third row and fourth column of a table has a certain value.

The tests within the registration tab do not cover the whole functionality. Nevertheless, the main use cases are implemented. The assertions are partly done with the values of GUI elements within the application others use the database entries to verify a user's action.

AddNewPrintJobArticleNum()
Within this test the first six digits of an article number are inserted to limit the amount of suitable articles to eighty. The button to open the articles' list becomes active immediately and is clicked. The fourth row is selected and confirmed, whereby the article is then displayed in a table. In the next step the reason for the print is selected from a combo-box and the print job is sent to the database by clicking the OK-button.
The assertion whether the new print job was inserted could be done in the label printing as well because after a maximum of ten seconds, the new print job has to be displayed in the overview area. However, if there would occur, for example, a failure when fetching new print jobs from the database, this test would fail, although the print job insertion was successful. Thus, and also to save the ten seconds idle time, the verification is done by the data stored within the database, see Listing 4.11. There must be a new entry in the Group table of type *9 - RePrintJob* and exactly one according entry (independent of the amount of labels to print) in the GroupItem table, inserted within the last

minute.

```
public void AddNewPrintJobMissingReason()
{
  #region Variable Declarations
  WinEdit uIItemEdit = this.UIReprintWindow.
     UITbArticleNumberWindow1.UIItemWindow.UIItemEdit;
  UITestControl uITbArticleNumberEdit = this.
     UIReprintWindow.UIPnlTopWindow.
     UITbArticleNumberWindow.UITbArticleNumberEdit;

  [...]
  #endregion

  // Type '12302' in text box
  uIItemEdit.Text = this.
     AddNewPrintJobMissingReasonParams.UIItemEditText;

  // Wait for 1 seconds for user delay between actions
  Playback.Wait(1000);

  // Click 'tbArticleNumber' text box
  Mouse.Click(uITbArticleNumberEdit, new Point(266, 16));

  [...]
}

[GeneratedCode("Coded UITest Builder", "11.0.60315.1")]
public class AddNewPrintJobMissingReasonParams
{
  #region Fields
  /// <summary>
  /// Type '12302' in text box
  /// </summary>
  public string UIItemEditText = "12302";
  #endregion
}
```

Listing 4.10: A code snip from a generated Coded UI test

AddNewPrintJobMissingReason()
This test does the same as the test above (AddNewPrintJobArticleNum) but does not select a reason from the combo-box. This leads to a pop up window when the OK-button is clicked.
In this case, the assertion of the test is done by verifying whether the GUI element (the pop up) exists.

```
private void validatePrintJobInsertion()
{
  detego.Service.DAL.SurveyorDataStoreEntities entities =
       new Service.DAL.SurveyorDataStoreEntities();
  TimeSpan timespan = new TimeSpan(0, 1, 0);
  var group = entities.Groups.Where(g => g.
     TimeStampCreated.Value.CompareTo(DateTime.UtcNow.
     Subtract(timespan)) > 0 && g.GroupType == 9).Take(1)
     ;
  Assert.IsFalse(group.FirstOrDefault().Processed.
     HasValue);

  var groupItem = entities.GroupItems.Where(gi => gi.
     GroupId == group.FirstOrDefault().GroupId);

  Assert.AreEqual(1, groupItem.Count());
  Assert.IsNotNull((groupItem.FirstOrDefault() as
     GroupItemProduct).ProductId);
}
```

Listing 4.11: Verification of the new print job

AddNewPrintJobGtin()
By using a GTIN, the article can be identified directly and must not be selected from a list. This test inserts a GTIN, adds the resulting print job by clicking on the Add-button and verifies the insertion in the same manner the AddNew-PrintJobArticleNum() test (Listing 4.11) does.

AddNewPrintJobBarcode()
This test simulates the barcode scan of a GTIN. The barcode scan is inserted via the keyboard buffer. As already mentioned, the application distinguishes between a manual insertion and a barcode scan by the delay between the digits and their amount. A short delay of below 100 ms and a series with at least eight digits is evaluated as barcode scan. In this case, and of course only if the GTIN is valid, the article is directly added to the list. After selecting a reason and sending the print job to the database, the assertion is done the same way as for the other tests.

PrintToXMLValid()
The purpose of this test was to start the print of a label. Instead of a real label the printer should write into a XML-file, which could be parsed and verified afterwards. However, it turned out that the button to start the print could not be executed automatically. This problem is described in more detail within Section 4.4.3.

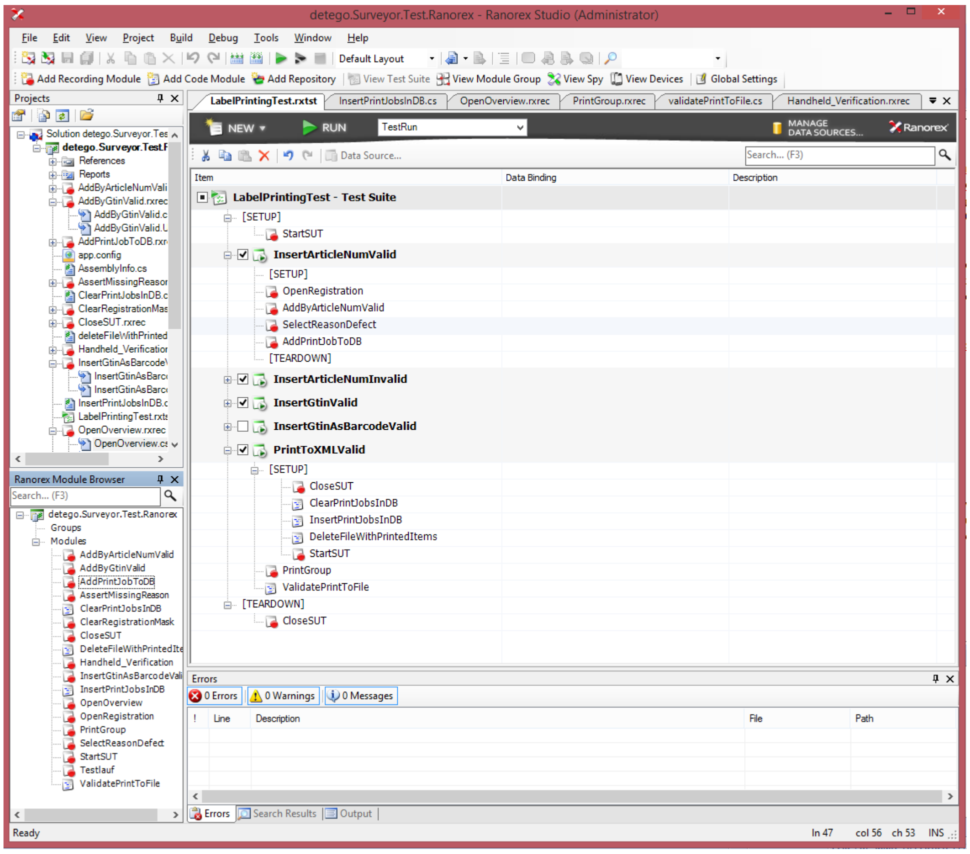
4.4.2.2 Ranorex framework

The Ranorex framework generates C# or VB code from recorded sessions, as an example see Listing 4.12. This code may be edited or expanded within the Ranorex Studio itself or otherwise Ranorex may be integrated into Visual Studio to write the automation code. Within this master's thesis the Ranorex environment has been used. Every recorded sequence (module) results in two code files (C# in this case): one with the generated code, which might be overwritten in time, and another one for the user specific code. Modifications within the latter will never be overwritten automatically. Again, any GUI element might be evaluated, similar to Coded UI tests.

The Ranorex framework provides a GUI for the test recordings. A recording results in a module, which may be expanded or split up into smaller ones afterwards easily. Additionally, within the GUI, the actions of a module may be changed, for example, the mouse click position or the entered value of a text box. Written user code may be called from there too, by adding a code module to a test or a user code action to a recording module.
All tests are organized within a Ranorex solution file. Each project within the solution represents a test suite. A suite, as well as every test case, has a set up and a tear down phase. These phases, equally like a test, may contain one or more modules. Figure 4.3 shows a test suite including some tests with recording and code modules, partly within the set up and tear down phases.
A recording module can only be changed within the GUI, not within the generated code. Figure 4.4 shows an example to change a mouse click event.

For these tests, the set up phase of the whole test suite is used to start the desktop application with the label printing module, respectively to close the application in the tear down phase. Only for some tests the application must be closed and started additionally within the test's set up phase. This is necessary because in order to execute the test correctly, the data within the database has to be modified. Any potential open print jobs are cancelled and a specific new print job is created. Since, these data are only loaded at the application start and not at runtime, a restart is inevitable. Generally, the application could be started and closed in each test case's set up and tear down phase. The disadvantage in this case is the unnecessary time consumption of a few seconds for each test case.

The implemented test cases are similar to the Coded UI tests mentioned above. The previously written C# code for the Coded UI tests concerning the assertions of the data in the database and the creation of any needed print job have been reused completely for the Ranorex tests.
Additionally, as there was no problem to start a print job within this framework, the print execution has been tested too. In the following, the tests explained

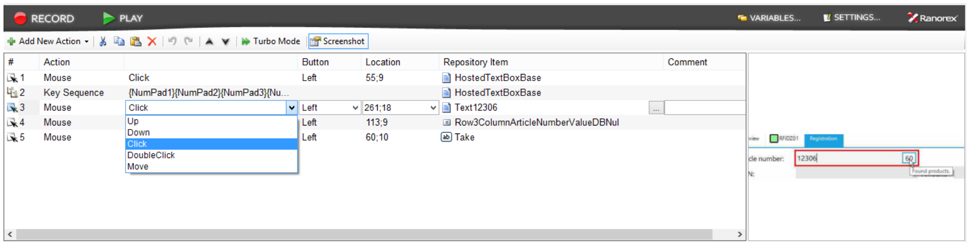


```
LabelPrintingTest - Test Suite
  [SETUP]
    StartSUT
  ☑ InsertArticleNumValid
    [SETUP]
    OpenRegistration
    AddByArticleNumValid
    SelectReasonDefect
    AddPrintJobToDB
    [TEARDOWN]
  ☑ InsertArticleNumInvalid
  ☑ InsertGtinValid
  ☐ InsertGtinAsBarcodeValid
  ☑ PrintToXMLValid
    [SETUP]
      CloseSUT
      ClearPrintJobsInDB
      InsertPrintJobsInDB
      DeleteFileWithPrintedItems
      StartSUT
    PrintGroup
    ValidatePrintToFile
  [TEARDOWN]
    CloseSUT
```

```
Ranorex Module Browser
detego.Surveyor.Test.Ranorex
  Groups
  Modules
    AddByArticleNumValid
    AddByGtinValid
    AddPrintJobToDB
    AssertMissingReason
    ClearPrintJobsInDB
    ClearRegistrationMask
    CloseSUT
    DeleteFileWithPrintedIte
    Handheld_Verification
    InsertGtinAsBarcodeVali
    InsertPrintJobsInDB
    OpenOverview
    OpenRegistration
    PrintGroup
    SelectReasonDefect
    StartSUT
    Testlauf
    ValidatePrintToFile
```

Errors: 0 Errors, 0 Warnings, 0 Messages

Figure 4.3: Test Suite within the Ranorex Studio



#	Action		Button	Location	Repository Item	Comment
1	Mouse	Click	Left	55;9	HostedTextBoxBase	
2	Key Sequence	{NumPad1}{NumPad2}{NumPad3}{Nu...			HostedTextBoxBase	
3	Mouse	Click	Left	261;18	Text12306	
4	Mouse		Left	113;9	Row3ColumnArticleNumberValueDBNul	
5	Mouse		Left	60;10	Take	

Figure 4.4: Recording module to add a print job within the Ranorex Studio

above within the Coded UI section are not mentioned again. They had been implemented within the Ranorex framework too, but the logic behind is more or less the same.

PrintToXMLValid()
This test closes the SUT, cancels all open print jobs within the database and inserts a new one for a specific article in the set up phase, to assure that the right article will be printed. In case of an already existing file with data from an elapsed print, this file will be deleted. Additionally, before the SUT is started again, some properties (the file path and a boolean) in a XML-file are changed in order to write the data into the file (see an example in appendix B) instead

of onto a label.
The test starts the print process by clicking a specific button and changes to the printer's area. First, the successful print is verified within the GUI. Next, the XML file with the printer's output is parsed and the resulting values are evaluated. As the label to print is known and always the same, the output is compared with hard coded reference values. In the future, if, for some reasons, the printed article should not be the same each time, these reference values could also be taken from the database dynamically.

```
[System.CodeDom.Compiler.GeneratedCode("Ranorex", "5.2.0")]
void ITestModule.Run()
{
  Mouse.DefaultMoveTime = 300;
  Keyboard.DefaultKeyPressTime = 100;
  Delay.SpeedFactor = 1.0;

  Init();

  Report.Log(ReportLevel.Info, "Mouse", "Mouse Left Click item 'MainForm.Print.HostedTextBoxBase' at 55;9.", repo.MainForm.Print.HostedTextBoxBaseInfo, new RecordItemIndex(0));
  repo.MainForm.Print.HostedTextBoxBase.Click("55;9");
  Delay.Milliseconds(200);

  Report.Log(ReportLevel.Info, "Keyboard", "Key sequence '{NumPad1}{NumPad2}{NumPad3}{NumPad0}{NumPad6}' with focus on 'MainForm.Print.HostedTextBoxBase'.", repo.MainForm.Print.HostedTextBoxBaseInfo, new RecordItemIndex(1));
  repo.MainForm.Print.HostedTextBoxBase.PressKeys("{NumPad1}{NumPad2}{NumPad3}{NumPad0}{NumPad6}");
  Delay.Milliseconds(0);

  Report.Log(ReportLevel.Info, "Mouse", "Mouse Left Click item 'MainForm.Print.Text12306' at 261;18.", repo.MainForm.Print.Text12306Info, new RecordItemIndex(2));
  repo.MainForm.Print.Text12306.Click("261;18");
  Delay.Milliseconds(200);

  Report.Log(ReportLevel.Info, "Mouse", "Mouse Left Click item 'InfoForm.Row3ColumnArticleNumberValueDBNul' at 113;9.", repo.InfoForm.
    Row3ColumnArticleNumberValueDBNulInfo, new
    RecordItemIndex(3));
  repo.InfoForm.Row3ColumnArticleNumberValueDBNul.Click("
    113;9");
  Delay.Milliseconds(200);

  Report.Log(ReportLevel.Info, "Mouse", "Mouse Left Click
     item 'InfoForm.Take' at 60;10.", repo.InfoForm.
    TakeInfo, new RecordItemIndex(4));
  repo.InfoForm.Take.Click("60;10");
  Delay.Milliseconds(200);
}
```

Listing 4.12: Resulting code of module in Figure 4.4

PrintErrorValid()
The print must be prepared in the same way as described in the test above. The properties within the XML file are modified, such that instead of printing to a file, a printer simulator is used. The simulator has an error probability, which refers on a printer error while the print. This probability must be set in a first step from 0 to 100 percent, to assure a print error occurs in the next print.
After executing the print job, the state of the desktop application is evaluated and the test verifies whether the print error is shown correctly. Usually, a print error (for example a paper jam) has to be resolved at the printer itself. The application resumes the print, when the printer sends the signals to be ready again. Therefore, after the evaluation of the error, the error probability of the simulator is set back to 0 percent to verify that the printer gets into the ready state and the print continues. Finally, the successful print is verified within the GUI.

4.4.3 Problems

Two types of issues occurred while working with *Coded UI* tests.
On the one side, the default configuration of Coded UI caused the test AddNew-PrintJobGtin() to fail. The test treated the input as the GTIN barcode scan feature mentioned in Chapter 4.4.1 and the Add-Button did not become clickable while executing the test. By default a replayed test inserts the number with a short delay between the digits, and the application interpreted the insertion as a barcode read. To fix this, the delay was set to 101 ms. Nevertheless, this behavior has been used within the AddNewPrintJobBarcode() test to verify the feature.
On the other side, working with the Coded UI framework has been tedious at times. The test sequence, once generated, cannot be changed easily. The recorded actions are mapped exactly to this element (for example the cell in

the first row and third column of a table) and changing the element afterwards is time consuming. Therefore, in the purpose of the maintainability it is necessary to record and generate short test sequences, which might be replaced in the future if the behavior of the application changes.
One bug regarding the delay time between actions exists in the framework. Due to load times, a short delay has to be simulated at some points too. Otherwise the needed data are not available in the application. The test generates such a delay as:

```
// Wait for 1 seconds for user delay between actions
Playback.Wait(10);
```

The Playback.Wait()-method expects an integer describing the ”milliseconds to wait” as argument. As the example above the comment describes a delay of one second, but just 10 instead of 1000 milliseconds are handed over, the test always fails. This implies that whenever a delay is necessary, the test has to be adapted manually after the generation. The problem is that a new generation of any test case re-generates each Coded UI test. Therefore, changes made to the generated code will be probably overwritten over time, which makes a bug, like the one just described, annoying.

As the last, and probably most important issue, one button within a table necessary to start the print job execution, could not be found by the test framework and therefore could not be clicked automatically whenever the test was replayed. It could not be figured out how to recognize this button within this test framework, even by changing the application or testing code. As this is the only possibility to start a print job execution within the application, and the Coded UI framework seems to not support an event invocation, the print itself could not be tested.

In contrast, there have not been any problems with the *Ranorex* framework to test the label printing module of the desktop application.

4.4.4 Results

Due to the problems described above, the Coded UI framework has been evaluated as not practicable within this master's thesis. The delay function is needed in nearly every test case. Additionally, the print function, which is the major feature of the module, cannot be tested, at least not at the moment.

In contrary no major issues occurred using *Ranorex* framework. The tests could be generated easily and if the tests are split up into small parts after the generation, a modification after some time (i.e. exchanging a button for any reason) or reusing those parts within another test may be done easily. Additionally, this framework is very intuitive to work with, and the company provides a helpful documentation.

4.5 Mobile Device Application

The mobile device tests in this context have been kept to a minimum. The purpose is to perform an inventory of one transponder automatically.

4.5.1 Preliminaries and Prerequisites

Some properties within two XML-files must be set before deploying the package to the mobile device. This assures that the application tries to connect to an existing and running web service, and will be updated automatically whenever a new version is available. This is a feature, which may be used for regression tests in the future. A nightly version may be deployed automatically in this way. In detail, within the entities.xml the following properties must be configured:

- <entity name="General.DataWebServiceUrl" value= "http://<hostname>/detego.Service/DataService.svc/Json" [...]/>
- <entity name="General.ReportServiceUrl" value= "http://<hostname>/detego.Service/ReportServiceOdata.svc/" [...]/>
- <entity name="General.OdataServiceUrl" value= "http://<hostname>/detego.Service/DataServiceOdata.svc/" [...]/>
- <entity name="General.AuthenticationWebServiceUrl" value= "http://<hostname>/detego.Service/AuthenticationService.svc/Json" [...]/>
- <entity name="General.ConfigurationWebServiceUrl" value= "http://<hostname>/detego.Service/ConfigurationService.svc/Json" [...]/>
- <entity name="General.UpdateManagerActive" value="true" [...]/>

Thereby <hostname> has to be replaced by the host of the server. Additionally, within the applications.xml the following parameters are necessary:

- <Username>sp</Username>
- <Password>sp</Password>
- <TestMode>true</TestMode>

To deploy the package and later on to perform the test, the mobile device must be connected to the network. Of course, as for all the other tests, the service and database specified as host within the entities.xml must be running and a transponder must be placed within the reader's field. In order to save the inventory run to the database, the EPC of this transponder must correspond to an article existing in the database.

The system has already a deploy mechanism implemented which, if set to true, compares the deployed version with the actual one within the database. Whenever the versions are different, the application is going to be deployed and automatically started.

4.5.2 Test Implementation

Since all evaluated tools for GUI testing do not support (remote) testing of mobile applications, a dedicated test-mode was implemented within the mobile application. This causes the application to login automatically, start the process ”Inventory Registration” and select the location ”Salesfloor”. By design, the application does not activate the RFID-reader automatically. This must be done manually via the trigger-button (hardware) on the mobile device and can not be performed through the touch screen.
Therefore, in every case, the activation of the RFID-reader must be done within the source code:

```
if (ControlLogic.GetInstance().TestMode)
{
  Tools.Instance().RfidReader.Activate();
}
```

The test was implemented as such, that the RFID-reader gets deactivated after at least one transponder has been read. Due to a high reading rate more than one transponders might be read, which does not matter. The next step is to send the read data to the web service/database, which is done by clicking the OK-Button within the application. The application changes automatically back to the location selection screen. The verification of the test occurs on the mobile device itself. After performing the inventory registration, the database is queried for a group entry with the given timestamp and verifies that such a group exists. According to the result an info or error message is logged. Therefore a new class had been implemented, see Listing 4.13. This verification may be expanded to verify the location and group type if necessary in future. The test has to navigate back to the main screen, in order to bring the application into a state, where the automatic update mechanism has the rights to deploy a newer software version in case. The test does not start again from the main menu, except the module is started again for example by hand. In this case the test would start again and perform the whole test run.

```
public class TestVerification
{
    #region Constructors
    private TestVerification()
    {
    }
    #endregion

    #region Fields
    private static readonly ILog logger = LogManager.
       GetLogger(typeof(TestVerification));
    #endregion

    #region Methods
    public static void VerifyInventoryRegistration(
       DateTime datetime)
    {
        Group group = Tools.Instance().
           GetGroupByTimeStampEvent(datetime);

        if (Tools.Instance().GetGroupByTimeStampEvent(
           datetime) == null)
            logger.Error("Error automated test: Verifing
               Inventory Registration failed");
        else
            logger.Info("Info automated test: Verifing
               Inventory Registration ok");
    }
    #endregion
}
```

Listing 4.13: Test class to verify the inventory registration

4.5.3 Problems

The main problem occurring at the beginning, was the fact, that the test could not be started with a GUI test and that the RFID-reader could only be activated from a hardware trigger. The solution to both issues had been to introduce the property ”TestMode” and program the whole test within the application itself. Within the application code it is possible to activate the RFID-reader. This affected also the test verification, hence, in this case, it must be done on the mobile device as well instead of doing it within the GUI test, as thought before.

4.5.4 Results

Whereas the integration of dedicated hardware in the test-loop poses some challenges, the overall test method proofs to be highly efficient and easily manageable. By using the described approach of a dedicated test-mode, basic functions can be repeatedly tested in an automated way which is an important cornerstone for regression tests within the software development cycle.

Chapter 5

Conclusion

This master's thesis proposes a feasible way to automate tests for multi-layered and distributed software frameworks using RFID. The basis for this work builds the software system detego® SURVEYOR, developed by the Enso Detego GmbH. In a conceptual run the possibilities in order to test this system automatically have been evaluated. Within the practical work of this master's thesis several additional challenges occurred, which partly could be solved.

The *Web service* has been tested with unit tests. Those tests are now usable for regression tests too. Thereby no major problems occurred. As a future work, it might be considered to implement integration tests for the web service and test several units in a specific dependency. Especially those tests concerning the database views might be expanded further.

The test use cases for the *Web application* have been divided into unit and functional GUI tests, depending on the test's purpose.
Several issues were encountered concerning the tests developed with the QUnit framework in the beginning. Since the implemented tests would be also usable for regression tests, one future work will be the integration into an automatic build process within the TFS. A viable solution for this might be the open source JS Test Runner Chutzpah[1]. Moreover, since most of the tests are the same for each report, a lot of effort could be avoided if the scripts of the web pages could be loaded automatically into the tests, each after the last one terminated. The test results of a web page should not be overridden by the next results. A quick solution was not found and finding one solution for both issues is out of scope for this master's thesis.
The framework used for the functional GUI tests was only usable to test simple operations. There might be further improvements of this framework, such that the tests could be expanded in the future. As alternative the *Ranorex* framework could be evaluated for these tests too.

For the *Desktop application* several test frameworks have been evaluated as unusable until the tests could be realized with the Ranorex test framework. It is a future task to develop this kind of tests for the remaining modules

[1]Last retrieved on 06.05.2015 from http://blogs.msdn.com/b/visualstudioalm/archive/2012/07/09/javascript-unit-tests-on-team-foundation-service-with-chutzpah.aspx

of the desktop application, which were not covered within this master's thesis.

The first try to use a functional GUI test for the *mobile device application* was not applicable. Instead, an integration test has been implemented on the device itself, which starts if a specific property is set. Since this works smoothly, it would be possible to write such tests for other modules of the application too. As a consequence, this is only useful within an continuous integration test, as the evaluation has to be modified, such that the test result is not only display within the log file of the mobile device.

Concluding, although there occurred several issues, some of the frameworks were practically not usable and there are still opportunities to improve the tests further, they do reduce the manual testing effort significantly.

Bibliography

Paul Baker, ZhenRu Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. User-interface testing. In *Model-Driven Testing*, pages 117–124. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-72562-6. doi: 10.1007/978-3-540-72563-3_9.

John E. Benetley, Wachovia Bank, and Charlotte NC. Software testing fundamentals - concepts, roles, and terminology. *SUGI 30*, Apr 2005.

Pakinam N. Boghdady, Nagawa L. Badr, Mohamed Hashem, and Mohamed F. Tolba. Test case generation and test data extraction techniques. *International Journal of Electrical & Computer Sciences IJECS-IJENS*, 11(03):82–89, June 2011.

David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. W3c note, World Wide Web Consortium, February 2004. URL `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211`.

J. Burnim and Sen Koushik. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 443–446, Sept 2008. doi: 10.1109/ASE.2008.69.

Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. W3c note, World Wide Web Consortium, March 2001. URL `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

E. Collins, A. Dias-Neto, and V.F. de Lucena. Strategies for agile software testing automation: An industrial experience. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pages 440–445, July 2012. doi: 10.1109/COMPSACW.2012.84.

Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15 (10):859–866, October 1972. doi: 10.1145/355604.361591. 1972 ACM Turing Award Lecture.

John Dooley. *Software Development and Professional Practice*. Apress, Berkely, CA, USA, 1st edition, 2011. ISBN 1430238011, 9781430238010.

Joe W. Duran and S.C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438–444, July 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010257.

Elfriede Dustin. *Effective Software Testing: 50 Ways to Improve Your Software Testing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201794292.

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

M E Eagan. Advances in software inspections. *IEEE Trans. Softw. Eng.*, 12 (7):744–751, July 1986. ISSN 0098-5589.

Klaus Finkenzeller. *RFID Handbook: FUNDAMENTALS AND APPLICATIONS IN CONTACTLESS SMART CARDS, RADIO FREQUENCY IDENTIFICATION AND NEAR-FIELD COMMUNICATION*. John Wiley & Sons, Inc., 3 edition, 2010. ISBN 0470695064.

Gordon Fraser. *Automated Software Testing with Model Checkers*. dissertation, Graz University of Technology, October 2007. URL `http://www.ist.tugraz.at/staff/fraser/papers/dissertation.pdf`.

Gordon Fraser and Franz Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In *Proceedings of the International Conference on Software Engineering Advances*, ICSEA '07, page 45, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2937-2. doi: 10.1109/ICSEA.2007.71.

Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: A survey. *Journal for Software Testing, Verification and Reliability*, 19(3):215–261, September 2009. ISSN 0960-0833. doi: 10.1002/stvr.v19:3.

Mats Grindal and Brigitta Lindström. Challenges in testing real-time systems. In *Proceedings of the 10th International Conference on Software Testing Analysis and Review*, EuroSTAR '02, 2002.

W.J. Gutjahr. Partition testing vs. random testing: the influence of uncertainty. *Software Engineering, IEEE Transactions on*, 25(5):661–674, Sep 1999. ISSN 0098-5589. doi: 10.1109/32.815325.

Dick Hamlet. When only random testing will do. In *Proceedings of the 1st*

International Workshop on Random Testing, RT '06, pages 1–9, New York, NY, USA, 2006. ACM. ISBN 1-59593-457-X. doi: 10.1145/1145735.1145737.

Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

Bill Hetzel. *The Complete Guide to Software Testing.* QED Information Sciences, Inc., Wellesley, MA, USA, 2nd edition, 1988. ISBN 0894352423, 9780471565673.

IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, December 1990.

Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Random vs. structure-based testing of answer-set programs: An experimental comparison. In JamesP. Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Computer Science*, pages 242–247. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-20894-2. doi: 10.1007/978-3-642-20895-9_26.

Y. Labiche, P. Thévenod-Fosse, H. Waeselynck, and M.-H. Durand. Testing levels for object-oriented software. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 136–145, New York, NY, USA, 2000. ACM. ISBN 15811320692. doi: 10.1145/337180.337197.

N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940.

Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA'06, pages 376–390, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35909-5, 978-3-540-35909-8. doi: 10.1007/11787044_28. URL `http://dx.doi.org/10.1007/11787044_28`.

Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* Wiley Publishing, 3rd edition, 2011. ISBN 1118031962, 9781118031964.

Simeon Ntafos. On random and partition testing. *SIGSOFT Softw. Eng. Notes*, 23(2):42–48, March 1998. ISSN 0163-5948. doi: 10.1145/271775.271785.

Roy Osherove. *The Art of Unit Testing: With Examples in .Net.* Manning

Publications Co., Greenwich, CT, USA, 1st edition, 2009. ISBN 1933988274, 9781933988276.

Jiantao Pan. Software testing. Technical report, Carnegie Mellon University, 1999. http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/.

A. Ruiz and Y.W. Price. Test-driven gui development with testng and abbot. *Software, IEEE*, 24(3):51–57, May 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.92.

Ben H. Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, November 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.031. URL `http://dx.doi.org/10.1016/j.jss.2009.06.031`.

Software Testing Fundamentals. Last retrieved on 03.02.2014 from http://softwaretestingfundamentals.com/, Dec 2010.

S. Sumathi and S. Esakkirajan. Structured query language. In *Fundamentals of Relational Database Management Systems*, volume 47 of *Studies in Computational Intelligence*, pages 111–212. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-48397-7. doi: 10.1007/978-3-540-48399-1_4. URL `http://dx.doi.org/10.1007/978-3-540-48399-1_4`.

Gerrit Tamm and Christoph Tribowski. *RFID*. Informatik im Fokus. Springer, 2010. ISBN 9783642114601.

Nikolai Tillmann and Jonathan Halleux. Pex–white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-79123-2. doi: 10.1007/978-3-540-79124-9_10.

J.A. Whittaker. What is software testing? and why is it so hard? *Software, IEEE*, 17(1):70–79, Jan 2000. ISSN 0740-7459. doi: 10.1109/52.819971.

T. Wissink and C. Amaro. Successful test automation for software maintenance. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 265–266, Sept 2006. doi: 10.1109/ICSM.2006.63.

Xiaochun Zhu, Bo Zhou, Juefeng Li, and Qiu Gao. A test automation solution on gui functional test. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 1413–1418, July 2008. doi: 10.1109/INDIN.2008.4618325.

List of Abbreviations

Term	Description
AJAX	Asynchronous JavaScript and XML
Auto-ID	Automatic Identification
API	Application Programming Interface
DBMS	Database Management System
DOM	Document Object Model
EPC	Electronic Product Code
GUI	Graphical User Interface
GTIN	Global Trade Item Number
HH	RFID-Handheld
HTML	Hypertext Markup Language
IIS	Internet Information Service
JS	JavaScript
JSON	JavaScript Object Notation
LF	Low Frequency
HF	High Frequency
HTTP	Hypertext Transfer Protocol
OData	Open Data Protocol
OO	Object-Oriented
OS	Operating System
REST	Representational State Transfer
RFID	Radio Frequency Identification

SC	Stockroom Coordinator
SHF	Super High Frequency
SOAP	Simple Object Access Protocol
SP	Sales Person
SQL	Structured Query Language
SUT	System Under Test
SVN	Apache Subversion
TFS	Microsoft Team Foundation Server
TFVC	Team Foundation Version Control
TTF	Telerik Testing Framework
UHF	Ultra High Frequency
WCF	Windows Communication Foundation
WPF	Windows Presentation Foundation
WSDL	Web Services Description Language
WWS	Warenwirtschaftssystem
XML	Extensible Markup Language

List of Figures

List of Tables

Appendix A

Requirements from Specification Sheets

Receipt of goods (complete) - mobile device (Handheld) application

Actors	Sales Person (SP), Stockroom Coordinator (SC), RFID-System, Warenwirtschaftssystem (WWS)
Pre-conditions	RFID-Handheld (HH) ist eingeschaltet und aufgeladen. Lieferscheine wurden elektronisch von WWS an das RFID System übermittelt. Die Artikelstammdaten der Warensendung sind dem RFID System bekannt. Sind keine Artikelstammdaten bekannt, werden die Artikel ignoriert. Aufrechte Verbindung zu WWS zur Übertragung der RECADVs vom zentralen RFID System. Die Warensendung ist vollständig. (alle Kartons und alle EPCs sind vorhanden)
Success end condition	Alle Kartons und deren Inhalt wurden vollständig erfasst. RECADVs wurde an WWS zurückgemeldet.

Main Success Scenario

1. SP/SC nimmt die Warensendung an.
2. SP/SC startet das Modul "'Wareneingang"' am HH.
3. SP/SC bringt Ware zum Warenerfassungsort.
   Kann dies in der Filiale sowohl eine "'Verkaufsfläche"' als auch ein "'Nachschublager"' sein, wählt der SP/SC den entsprechenden Warenerfassungsort aus. Gibt es nur eine Möglichkeit, wird dieser Warenerfassungsort voreingestellt und muss nicht händisch ausgewählt werden.
4. SP/SC drückt die Scan Taste am HH, um den Scan Vorgang zu starten.

5. Am HH werden alle erfassten EPCs automatisch den SSCCs zugeordnet und farblich markiert dargestellt (Listendarstellung).

6. SP/SC scannt solange bis kein akustisches Signal mehr hörbar ist. SP/SC stellt per Sichtkontrolle fest, dass die Anzahl der gescannten Lieferscheine/SSCCs mit der tatsächlich angelieferten Anzahl übereinstimmt und auch alle gelieferten Kartons für diese Filiale bestimmt sind.

7. SP/SC stellt fest, dass alle Zeilen grün sind und die Anzahl bei "'Unvollständig"' 0 ist.

8. SP/SC drückt "'Vollständige bestätigen"'. Die Liste am Handheld wird leer.

9. SP/SC drückt "Zurück" um in den Start-Screen zu wechseln.

Inventory with check against target list - mobile device (Handheld) application

Actors	SP, SC, RFID-System (Handheldapplikation), WWS
Pre-conditions	RFID-Handheld ist eingeschaltet und aufgeladen. Alle Transponder sind vorhanden und funktionsfähig.
Success end condition	Die Bestandsaufnahme(n) wurden erfolgreich durchgeführt.

Main Success Scenario

1. SP/SC startet das Modul "'Bestandsprüfung"'

2. SP/SC wählt aktiv die Lagerfläche aus, auf der eine Bestandsaufnahme erfolgen soll. Standardmäßig ist das Feld leer. Bleibt es leer kommt es zu einer Fehlermeldung.

3. SP/SC kann auf Division, Liefertermin, Warengruppe oder Artikelnummer oder GTIN, die Bestandsaufnahme einschränken. Bleiben die Felder leer, werden alle erfassten Artikel auch auf der Handheld Oberfläche angezeigt. (Anmerkung: im detego Data Store werden alle erfassten Artikel gespeichert – in der Reporting-Oberfläche kann dann wieder gefiltert werden.)

4. SP/SC klickt auf "'Weiter"'

5. SP/SC erhält eine Liste aller erwarteten, dem Filter entsprechenden Artikel. SP/SC scannt im Idealfall so lange bis alle Artikel grün dargestellt sind – d.h. "‘Soll"’ und "‘Ist"’ gleich sind.
6. SP/SC schließt die Bestandsaufnahme durch Drücken von "‘Bestätigen"’ ab und gelangt wieder in die Filter-Auswahl.
7. SP/SC wählt eine neue Lagerfläche und/oder Filter aus und wiederholt die Schritte 3 bis 7 bis keine Bestandsaufnahme mehr durchgeführt werden soll.
8. SP/SC drückt "‘Zurück"’ und gelangt in den Start-Screen.

Reprint with reference label - mobile device (Handheld) application

Actors	SP, SC, RFID-System (Handheldapplikation), WWS
Pre-conditions	RFID-Handheld ist eingeschaltet und aufgeladen. Artikelinformation wird anhand eines Referenzetiketts ermittelt. Größe und/oder Farbe sollen geändert werden.
Success end condition	Nachdruck-Etiketten wurden erfolgreich erfasst.

Main Success Scenario

1. SP/SC startet das Modul "‘Etikettennachdruck"’
2. SP/SC scannt den EAN Barcode des Referenz-Etiketts.
3. SP/SC drückt "‘Weiter"’
4. Der EAN mit all der ermittelten Artikelinformation wird angezeigt. SP/SC hakt "‘Größe/Farbe ändern"’ an.
5. SP/SC drückt auf "‘Weiter"’ und gelangt zur Größenauswahl.
6. SP/SC wählt auf die gewünschte Größe aus dem entsprechenden dropdown-Feld.
7. SP/SC wählt auf die gewünschte Farbe aus dem entsprechenden dropdown-Feld.
8. SP/SC drückt auf "‘Weiter"’ und gelangt in den Nachdruck-Screen.

9. SP/SC ändert gegebenenfalls die Anzahl der nachzudruckenden Etiketten für diesen EAN.

10. SP/SC drückt "‘Druckauftrag senden"’. Ein bereitgestelltes Webservice nimmt den Druckauftrag entgegen.

Print of the ordered label - Desktop application

Actors	SP, SC, RFID-System (Desktopapplikation - Etikettennachdruck), WWS
Pre-conditions	Aufrechte Verbindung zum WWS zur Übertragung der Druckaufträge. RFID Drucker angeschlossen, eingeschaltet und über Netzwerkverbindung erreichbar. RFID Drucker mit Farbband und RFID Transpondern bestückt.
Success end condition	RFID Etiketten sind korrekt, vollständig und eindeutig gedruckt. Generierte EPCs sind an das ERP-System zurückgemeldet.

Main Success Scenario

1. SP/SC startet das Modul "‘Etikettennachdruck"’ an der Nachdruckstation (PC).

2. In der Übersicht der GUI werden die ausstehenden Druckaufträge angezeigt.

3. SP/SC wählt gegebenenfalls einen Drucker (Default-Drucker standardmäßig ausgewählt) und drückt "‘Drucken"’.

4. RFID-System beginnt die Kommunkation mit dem RFID-Drucker, berechnet und sendet die zu druckenden Daten der einzelnen Etiketten. RFID-Drucker beginnt die Etiketten zu drucken, und gibt für jedes Etikett eine Erfolgsmeldung an das RFID-System zurück.
   In der Übersicht und der Detailansicht der GUI wird der Druckfortschritt angezeigt.

5. RFID-System schließt den Druck ab und meldet die generierten EPCs an das WWS zurück.

Error handling while printing - Desktop application

Actors	SP, SC, RFID-System (Desktopapplikation - Etikettennachdruck), WWS
Pre-conditions	Aufrechte Verbindung zum WWS zur Übertragung der Druckaufträge. RFID Drucker angeschlossen, eingeschaltet und über Netzwerkverbindung erreichbar. RFID Drucker mit Farbband und RFID Transpondern bestückt.
Success end condition	RFID Etiketten sind korrekt, vollständig und eindeutig gedruckt. Generierte EPCs sind an das ERP-System zurückgemeldet.

Main Success Scenario

1. SP/SC startet das Modul "'Etikettennachdruck"' an der Nachdruckstation (PC).

2. In der Übersicht der GUI werden die ausstehenden Druckaufträge angezeigt.

3. SP/SC wählt gegebenenfalls einen Drucker (Default-Drucker standardmäßig ausgewählt) und drückt "'Drucken"'.

4. RFID-System beginnt die Kommunkation mit dem RFID-Drucker, berechnet und sendet die zu druckenden Daten der einzelnen Etiketten. RFID-Drucker beginnt die Etiketten zu drucken, und gibt für jedes Etikett eine Erfolgsmeldung an das RFID-System zurück.
   In der Übersicht und der Detailansicht der GUI wird der Druckfortschritt angezeigt.

5. Etikettendruck wird durch einen Druckerfehler (leeres Farbband, leere Tag-Rolle, Papierstau) unterbrochen.

6. Der entsprechende Fehler wird am Drucker und in der GUI des Etikettennachdrucks angezeigt und fordert eine manuelle Interaktion.

7. SP/SC behebt den Fehler am Drucker (durch Ersetzen des Farbbandes bzw. der Tag-Rolle oder Beheben des Papierstaus) und quittiert gegebenenfalls den Fehler am Drucker und/oder in der GUI.

8. RFID-System fährt mit dem Drucken der Etiketten fort.

9. RFID-System schließt den Druck ab und meldet die generierten EPCs an das WWS zurück.

Appendix B

XML Printer Output

```
[...]
{PC000;0043,0074,1,1,J,00,B,P1=Size|}
{PC001;0145,0074,1,1,J,00,B=XS|}
{PC002;0043,0111,05,05,I,00,B,P1=|}
[...]
{PC009;0043,0332,05,05,I,00,B=Countries8|}
{PC010;0215,0111,05,05,J,00,B,P1=SEK|}
{PC011;0215,0142,05,05,J,00,B,P1=|}
[...]
{PC018;0366,0111,05,05,T,00,B,P3=1999,00|}
{PC019;0366,0143,05,05,T,00,B,P3=|}
[...]
{XB00;0045,0220,Q,20,08,05,0=3034F766CC009D401E28F2A2|}
{PC026;0205,0370,05,05,I,00,B,P2=empfohlener
    Verkaufspreis|}
{PC027;0205,0393,05,05,I,00,B,P2=recommended retail price
    |}
{PC028;0205,0414,05,05,I,00,B,P2=Prix conseilles|}
{PC029;0205,0458,1,1,T,00,B,P2=209501161505|}
{PC030;0050,0503,1,1,T,00,B=679|}
{XB01;0087,0523,5,3,03,0,0044,+0000000000,017,1,
    00=405342700629|}
{PC031;0205,0645,05,05,I,00,B,P2=506|}
{PC032;0135,0496,05,05,T,00,B=679|}
{PC033;0205,0668,05,05,I,00,B,P2=3034F766CC009D401E28F2A2
    |}
{XB02;0000,0000,r,P5,F0,T24,G2,R00000000,K00000000,L11111
    ,J00000000,V1|}
{RB02;3034F766CC009D401E28F2A2|}
{XS;I,0001,0003C3101|}
```

Listing B.1: Shortened output of a label print to file