



Anna Gertrud Haupt, BSc

Improving Quality of Customized Salesforce Implementations: Investigation on different Software Testing Methods

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

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

Institute of Software Technology

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

Graz, May 2022

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.

10.05.2022

Date

A handwritten signature in black ink, reading "Anna Haupt". The signature is written in a cursive style with a large, stylized 'A' at the end.

Signature

Abstract

Nowadays it becomes more and more important to bring software updates and new versions of a software product on the market. Therefore, it is essential that the maintainability and reliability of the software is always given.

In order to do this cost-effectively and also with as few effort as possible, value must be placed on the quality of the software at an early stage. It is therefore becoming increasingly important to place more value on the quality of a software, especially earlier in the application lifecycle.

The quality of the software can be significantly increased by using appropriate testing methods and a standard deployment process with additional quality assurance. Therefore, this master thesis focuses on investigating different testing methods in the context of Salesforce and analyzing their effectiveness. For this purpose, an empirical experiment is conducted to investigate which set of methods fits best during the different stages of the application lifecycle.

Furthermore, this work focuses on the elaboration of a suitable deployment strategy in the context of Salesforce implementations, including proposals for branching models and environment structures. The proposed strategies are based on a previously conducted analysis of the current development and deployment process of a company that specializes in Salesforce implementations.

Finally, this work deals with possible problems and solutions that have arisen during the implementation phase of these proposals at the mentioned company.

Keywords: Salesforce, Software Testing, Black-Box Testing, Development Process, Deployment Process, Application Lifecycle, Software Quality, Software Complexity, Gap Analysis

Kurzfassung

Heutzutage wird es immer wichtiger, Software-Updates und neue Versionen eines Softwareproduktes auf den Markt zu bringen. Daher ist es unerlässlich, dass die Wartbarkeit und Zuverlässigkeit der Software stets gegeben sind.

Um den Aufwand möglichst gering und die Kosten dabei minimal zu halten, muss im Software Entwicklungsprozesses frühzeitig Wert auf die Qualität der Software gelegt werden.

Durch den Einsatz geeigneter Testmethoden und eines standardisierten Deployment-Prozesses kann die Qualität der Software deutlich gesteigert werden. Daher liegt der Fokus dieser Masterarbeit, verschiedene Testmethoden im Kontext von Salesforce zu untersuchen und deren Effektivität zu analysieren. Zu diesem Zweck wurde ein empirisches Experiment durchgeführt, um herauszufinden, welche Methoden in den verschiedenen Phasen des Lebenszyklus einer Softwareanwendung am besten geeignet sind.

Darüber hinaus konzentriert sich diese Arbeit auf die Ausarbeitung einer geeigneten standardisierten Deployment- Strategie in Zusammenhang mit Salesforce Anwendungen. Des Weiteren werden Vorschläge für Git-Branching Modelle und einer einheitlichen Sandbox Struktur aufgezeigt. Die dargestellten Strategien basieren auf einer zuvor durchgeführten Analyse des aktuellen Entwicklungs- und Deployment-Prozesses eines Unternehmens, das sich auf Salesforce Implementierungen spezialisiert hat.

Abschließend befasst sich diese Arbeit mit Problemen und möglichen Lösungen, die während der Implementierungsphase dieser Vorschläge bei dem genannten Unternehmen aufgetreten sind und weitere Vorschläge um die Software Qualität zu steigern.

Stichwörter: Salesforce, Software Qualität, Software Komplexität, Funktionale Tests, Ist-Analyse, Softwareentwicklungsprozess

Contents

Abstract	iii
Kurzfassung	iv
1. Introduction	1
1.1. Background and Previous Work	1
1.2. Research Area	4
1.3. Problem Definition	5
1.4. Solution Approach	5
1.5. Scope and Limitations	5
2. Salesforce	7
2.1. Architecture	7
2.1.1. Salesforce Services	9
2.1.2. Salesforce Programming Technologies	10
2.2. Click-Based Development on Salesforce	11
2.3. Definition of different Roles and their Responsibilities	11
2.4. What are Salesforce specifics?	12
3. Different Environments	15
3.1. Sandbox Environments	15
4. Software Quality	18
4.1. Functional Quality	19
4.2. Structural Quality	19
4.3. Process Quality	20
5. Software Complexity	23
5.1. Code Complexity Metrics	23
5.2. Static Code Analysis	24
5.2.1. Useful Tools to calculate Cyclomatic Complexity	27
5.3. Limitations of Salesforce	27
6. Software Testing	28
6.1. Definition and the Goal of Software Testing	28
6.2. Software Testing Methods	29
6.2.1. Functional Tests	29
6.2.2. Non Functional Testing	30

6.2.3. Black & White-Box Testing	32
7. Gap Analysis of the current development process	35
7.1. Proposals to close gap	39
8. Empirical Experiment comparing different Software Testing Methods	40
8.1. Goal	40
8.2. Structure	41
8.2.1. Method	41
8.2.2. Selection of the Participants	42
8.2.3. Conducting the Examination	42
8.2.4. Preparation	46
8.2.5. Experimental Difficulties	47
8.3. Evaluation	48
8.3.1. Qualitative & Quantitative Results	49
8.3.2. Statistical Analysis	51
8.3.3. Analysis of Test Case Quality	54
8.4. Appropriate method depending on the Application Lifecycle Phase . .	57
8.4.1. Application Lifecycle	57
8.4.2. Relationship between evaluated Results and the Lifecycle Phase	58
9. Deployment Strategy	60
9.1. Git Workflow	60
9.1.1. Branching Strategy	61
9.2. Environment Structure	63
9.2.1. Refresh Lifecycle & Clean Up	65
9.2.2. Synchronization of Version Control and Environments	66
9.2.3. Continuous Integration	67
9.2.4. Load Sample Data	70
9.2.5. Deployment Frequency	71
9.2.6. Testing Strategies and Responsibilities	72
9.2.7. Considerations	72
10. Adaptions to the proposed Deployment Strategy	74
10.1. Synchronization between Version Control and Environments	74
10.2. Linting	75
10.3. Administration of Bug Reports	76
10.4. Introduction of Deployment Cycles	76
10.5. Further planned improvements	77
10.5.1. Creation of a common Error List	78
10.5.2. Quality Gate on Code Coverage	78
10.5.3. Integration of Git Commits with Salesforce Cases	78
10.5.4. Backup with Nightly Synchronization	78
10.5.5. Improve Requirements Engineering	79
10.5.6. Ensure quality of test cases with Mutation Testing	79

11. Conclusion	80
List of Abbreviations	84
Bibliography	85
Appendix A. Questionnaire	88
Appendix B. Test Setup	91
B.0.1. Task Descriptions	91

List of Figures

1.1.	Example level model for the area of code quality	3
1.2.	Left: First analysis Right: Analysis after the workshop	4
2.1.	Architecture of Single and Multi Tenant Applications	8
2.2.	Common CRM Systems illustrated distinguish by their architecture. . .	12
4.1.	Correlation between a late found error and costs.	19
5.1.	Correlation between LoC and CC in 3 Projects	25
5.2.	Left: Project 1. Right: Project 2 Bottom: Project 3	26
6.1.	Black-Box Testing explained	33
6.2.	White Box Testing explained	33
7.1.	Traditional quality model in contrast to the Shift left model	38
8.1.	Transition State Diagram of Invoice creation	46
8.2.	Spidergraph: Number of Errors per Method found	53
8.3.	5 Phases of the software development lifecycle	57
9.1.	GitFlow Branching Model	63
9.2.	Initial Organisations	63
9.3.	Deployment cycle of a new feature	65
9.4.	Deployment cycle of a hotfix	65
9.5.	Refresh Lifecycle of Sandboxes in combination with git branches	66
9.6.	Example GithubAction for Deployment Validation	68
9.7.	Github Action on merge Pull Request	69
9.8.	Correlation between Cost and Length of Feedback Cycle	71
B.1.	Screenflow for booking a vehicle	93
B.2.	Screenshot Task 1: Enter Booking Details	94
B.3.	Screenshot Task 1: Choose available Vehicle	94
B.4.	Screenshot Task 2: New Vehicle Creation	95

List of Tables

2.1. Difference between On Premise and Cloud Computing	7
3.1. Available Sandboxes per Salesforce Edition	17
6.1. Types of Testing	30
7.1. Gap analysis of the company's development process	36
8.1. Example of Decision Table - Invoice creation	43
8.2. Average number of test cases per method	51
8.3. Probabilities of error detection of the different methods in comparison .	52
8.4. Code Coverage of Random testing, Decision Table, Error Guessing, Transition State Diagram and the BVA method	55
8.5. Code Coverage of different types of BVA	56
B.1. Code Base Analysis	91

1. Introduction

This master thesis focuses on increasing the quality of customized Salesforce implementations by analyzing and evaluating different software testing methods. At the beginning of this thesis, the Salesforce ecosystem in general with special attention to the difference between the Salesforce architecture and the architecture of other common CRM Systems is explained. The software complexity of 3 different Salesforce implementations is also discussed in the chapter of [Software Complexity](#). Afterward, a gap analysis between the current development process and the desired process on a company that specializes in Salesforce implementation was conducted in terms of testing and code quality. This company has grown rapidly in a short period and therefore has more resources also more resources can be allocated to improving product quality.

The already performed project "Analysis and improvement of the Software development process of a StartUp company" which was carried out at an earlier time, serves as a basis for this work. The results of this project can be seen in the chapter [Background and Previous Work](#). To examine the effect of different testing methods on software quality, an empirical experiment is conducted in which different software testing methods were examined and evaluated. Software quality is divided into functional quality, structural or non-functional quality, and process quality. To increase the process quality, the standardization of the development process is concerned as well. This can be read in chapter [Deployment Strategy](#) and chapter [Adaptions to the proposed Deployment Strategy](#). Chapter [Conclusion](#) gives an overall summary of the thesis with suggestions for further improvements in software quality and potential Future work elaborated.

1.1. Background and Previous Work

Former studies were focused on analyzing and improving the development process of a software company. The company was founded in 2019 as a Start-Up company with headquarters in Graz. It is specialized in consulting and implementing customized Salesforce solutions. In the beginning, it was important for the company to deliver solutions as fast as possible. After 4 years and a significant increase in the number of customers and also in the employees, the company is more concerned about the quality regarding reusability, security, effectiveness, and maintenance of the systems. With only 2 developers at the beginning, it was not necessary to have a standardized

development process because every developer had his customer. The company has grown and now counts 25 people in total. Out of these 25 people, 15 are working on the development of customized Salesforce implementations. They are organized into 3 teams whereby a team consists of at least 2 developers and 2 Salesforce Administrators. Every team has its fixed assigned customers. The development process of each team differs as it is self-organized and there is no preset process defined.

As the company is growing constantly the leaders of the company realized that now it is time to focus more on the quality of the produced software and also the underlying development process.

The analysis from the project "Analysis and improvement of the Software development process of a StartUp company" is based on the so-called level model which was developed in cooperation with the Technical University of Graz and the FFG (Austrian Research Promotion Agency) in the workshop "Advanced Software Quality Assurance for Technical Management".¹

The level model relies on the definition of areas in which the company wants to analyze the development process. Once areas are defined, sub-items are determined, which must be present to perform well in the chosen area, regardless of whether they have already been implemented by the company or not. Then these sub-items are prioritized according to their importance. This results in a point scale that should be divided into 4 levels. The 4 levels indicate how well the company is positioned in the respective area. The levels range from level 1 which is not satisfying to level 4 which can be called expert level. The classification of the levels can be designed as desired depending on the company. At the end to execute the analysis, the total number of points is analyzed for each area, which then provides the corresponding level.

The definition of the areas, sub-items, and the scoring system which are used were discussed in the workshop between experts from the technical university of Graz and technical managers from software companies. Senior developers were also brought in and asked for their opinion. The participants all came from companies of different sizes and operation fields. Starting with a 2-person company up to a 350-person company in the fields of software development and consulting 9 companies took part in this workshop. An exact list of these companies and a detailed description of the workshop can be found on the homepage of TU graz.¹

For the research, the level model was applied with the categories of continuous integration, software testing, and software quality.

To get a better understanding of the level model the following Figure 1.1 shows an example of a level model configuration for the category of code quality.

This level model wants to improve in the category of code quality. The areas are defined as guidelines, review, source control, documentation, refactoring, knowledge

¹<https://www.tugraz.at/en/studying-and-teaching/degree-and-certificate-programmes/continuing-education/qualification-initiative/advanced-software-quality-assurance-for-technical-management/>



Figure 1.1.: Example level model for the area of code quality

sharing, and mindset. Every area has sub-items defined for example version control available and using feature branches. The shown level model has the restriction that in every column at least one point must be reached to reach the beginner's level. It is not sufficient enough if in one sub-item all points are reached and in another one no points are achieved at all.

As it is shown in Figure 1.2 on the left side the company reached level 1 in every category at the start of the workshop. That was the time when the company realized that it was urgently necessary to change something. It is very important not to try to implement too many changes at once as this can become very complicated and lead to many side effects.

Now that it was clear in which areas the company had the potential for improvement, the phase of brainstorming for improvements began. After this phase was done the company slowly started the implementation of the new concepts.

At the end of the workshop, the same analysis was conducted with a significant improvement at all levels. The graph on the right side of Figure 1.2 shows the distribution of the levels after the workshop was completed.

As seen in Figure 1.1 one restriction in the quality model was the use of version control but also the used level model of continuous integration had this restriction. Therefore, the company decided to address the introduction of version control as the first issue. This has led to a simultaneous increase in the areas of code quality and continuous integration.

Another improved modification was the introduction of code reviews and pair programming which led to an increase in the code quality and also had the benefit of learning effects. The use of pair programming and reviews has the advantage that less experienced programmers can learn from the senior developer's best practices. The senior developer can also show up common occurring errors which can be prevented in the future, which increases the code quality. The next considerable improvement

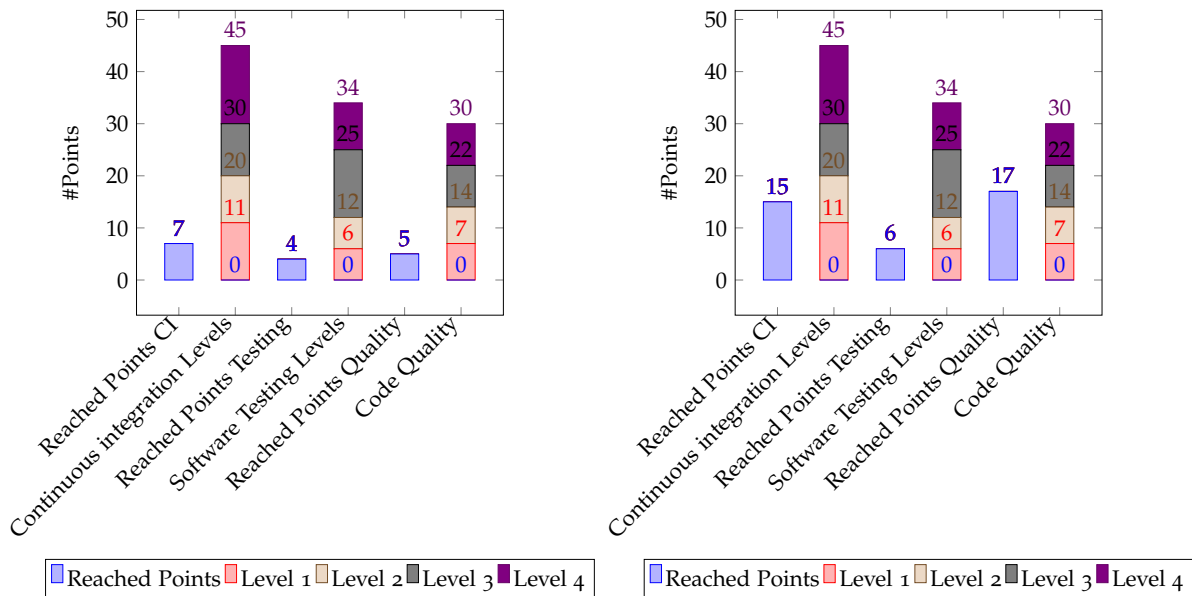


Figure 1.2.: Left: First analysis Right: Analysis after the workshop

was knowledge sharing and documentation which was not actively being used in the daily operation before. The least improvement was in the category of software testing. Therefore this is one more reason why this master thesis will focus on software testing which should also lead to an increase in software quality in the future. Despite this, it should be remembered that changes take time and the results and improvements are not always immediately visible. Of course, there were some problems with the conversions in the initial phase, but on the whole, it worked very well.

1.2. Research Area

This master thesis was initiated by me in collaboration with a small company based in Austria that is specialized in customized Salesforce Implementations. The purpose of this thesis is to provide a guideline for improving the quality of these customized Salesforce Implementations.

To achieve this an empirical experiment on different software testing methods with respect to customized Salesforce implementations is obtained. Besides this also the development process of the company should be improved.

During the prestudies, it was stated that the quality of testing also highly depends on a standardized development and testing process. Therefore, the focus is on the expansion to a standardized process in the company, which includes the standardized use of Version Control, the unique use of runtime environments as well as Continuous Integration and Delivery.

1.3. Problem Definition

The problem of this master thesis consists of two logically related parts. By answering the questions of these two problem areas, the purpose of this master thesis can be reached. The scope of the problem narrows as we move from a general view of software testing methods and deployment processes to a more Salesforce-specific one.

This leads to the formulation of the first problem:

“How can a standard software deployment process for Salesforce implementations be defined?”

Secondly, attention is paid to available software testing techniques:

“What are the standard software testing methods and which set of methods is best suited for the company?”

To finally introduce the described standard test process and test methods in the company, additional resources are required, especially in the initial phase.

The order of the next steps and which methods are implemented in the company is therefore a matter of the company's preferences. Some changes have already been implemented or are in progress which are described in Chapter 10. Chapter 11 will summarize this as well with several suggestions for the future.

1.4. Solution Approach

In order to be able to develop a suitable standard process, the current situation of the company must first be examined. For this reason, a comparison between the company's current development process and the desired one is carried out to determine possible deviations. After that, it has to be considered in which areas more time can be invested to improve them and which ones bring the most benefit.

Afterward with an empirical study different software testing methods were examined and analyzed to find out which one is the best fitting method concerning the application lifecycle.

1.5. Scope and Limitations

This thesis focuses more on what should be tested and how test cases can be defined and not on implementation details. Once it is clear what is to be tested, it becomes easier to write suitable test cases and to automate the execution later on.

The focus of this thesis is on Black-box Software Testing Methods instead of White Box Testing Methods. The difference between these two methods will be explained in Section 6.2.3. To expand this project with White Box Testing Methods could be a Future Work. More about this in the correlative Chapter 11 at the end of this thesis. It is important to understand that the proposed test methods only serve as a basis and guidance and cannot be considered as the only solution. There is not only one method that can be used for everything.

The proposed deployment process should serve as a guide and is only one way out of many to address this problem. It should be noted that it always depends on the particular needs and size of the company that wishes to implement such changes in order to improve their software quality. These suggested changes are based on the combination of the analysis of the level model and the gap analysis.

A well-defined requirement is a basis for fast and high-quality software development. Nevertheless, requirements engineering will not be part of this thesis as this would go beyond the scope.

2. Salesforce

Salesforce is a cloud-based company based in the US and was founded in 1999. Salesforce offers one of the most known customer relationship management (CRM) platforms with a focus on sales, customer service, marketing automation, and analytics. It also provides features to customize its inbuilt data structures and graphical user interface to suit the specific needs of a business.¹

Salesforce uses a cloud computing service. Table 2.1 shows the comparison between an on-premise architecture and a cloud computing architecture. **On-premises** means that the customer purchases the license and runs the application on their computers and servers.

In contrast to **cloud computing** where users subscribe to a software service that is hosted on the cloud provider's infrastructure.²

Function	On Premise	Cloud Computing
Software	Installed on company owned hardware resources	Delivered via service over Web
Access	Through application on computer	Through the internet
Upgrades	Company itself is responsible	Automatically provided by the cloud computing provider
Versions	Multiple versions to maintain	Single code base with no infrastructure to maintain
Hardware	Purchase, Maintain and Manage	Pay for what you need

Table 2.1.: Difference between On Premise and Cloud Computing
Source: Manchar and Chouhan (2017)

2.1. Architecture

The Salesforce architecture can be seen as a series of layers on top of each other. The core architecture of Salesforce can be divided into 3 layers which are multi-tenant, metadata, and API.

Multi tenant

An architecture where a single software instance can serve multiple users is called

¹Pullarao and Thirupathirao, 2013.

²Pullarao and Thirupathirao, 2013.

multi-tenancy.³

With a single-tenant, each user has their own application with its own associated database, whereas with a multi-tenant, everyone uses the same application but a separate database. Multi-tenant applications typically include some level of customization for the tenants, such as customizing the appearance of the application or allowing users to set specific access rights and restrictions.

By sharing resources, multi-tenancy tries to make it possible to achieve economic advantages in cloud computing. One of the biggest advantages of multi-tenancy architecture is the lowering of the costs from an individual user by sharing resources. Multi-Tenancy is seen differently from different service models. In the context of SaaS, multi-tenancy means that two or more customers utilize the same service or application provided by the cloud service provider (CSP) regardless of the underlying resources. In contrast to this, multi-tenancy in the context of IaaS occurs when two or more virtual machines belonging to different customers share the same physical machine. With IaaS the customer can control but not manage the underlying infrastructure.⁴ The use of available resources can be maximized by sharing machines with multiple tenants. Without multi-tenancy cloud computing services would be far less effective.

Figure 2.1 illustrates the difference between single tenant and multi tenant.

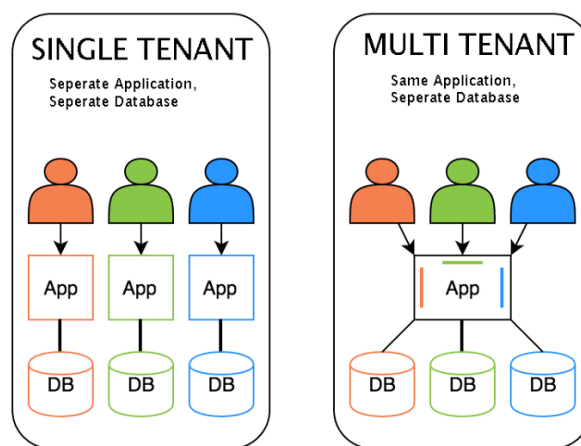


Figure 2.1.: Architecture of Single and Multi Tenant Applications.

Source: <https://medium.com/mendix/multi-tenancy-in-mendix-platform-part-i-463b8b892364>

Metadata

Metadata provides information about the structure and aspects of data. Metadata is used to facilitate data processing by summarizing basic information about the data.⁵ The Salesforce Platform uses a Metadata-driven development model. With Metadata it is possible to create and customize the structure of a user's Salesforce org.

³Krebs, Momm, and Kounev, 2012.

⁴AlJahdali et al., 2014.

⁵A *Guardian guide to metadata* 2013.

Application Programming Interface (API)

Salesforce provides some very powerful APIs that can simplify the connectivity and data communication between different devices and applications. Salesforce is following the "API first" approach. With this approach, the developer is more flexible in manipulating data. Salesforce data APIs are specifically designed for data manipulation in Salesforce, while others are more focused on analytics like the Analytics REST API.

2.1.1. Salesforce Services

Salesforce offers a variety of different services. Starting with software services, learning platforms, or even marketplaces for prefabricated applications. The section below will list some of these services with a brief description.

Trailhead

Trailhead is a free online learning platform developed by Salesforce that specializes in learning Salesforce technologies.⁶

It is possible to attend courses on all kinds of topics to obtain a Salesforce certificate afterward.

App Exchange

App Exchange is comparable to the Apple App Store or Google PlayStore. App Exchange offers customers the opportunity to install and purchase third-party applications or consulting services, in some cases even free of charge.

Software as a Service (SaaS)

Salesforce is served from a cloud. It runs centrally in the cloud in consequence it is not possible to build a local instance of Salesforce on your laptop. A specific Salesforce instance can be updated by deploying changes directly to these instances.

The Salesforce cloud solution includes a No-Code/Low-Code Platform. With this platform, developers can implement requirements faster, by using a graphical user interface with drag and drop options with reusable components. The so-called Process Builder or Flow Generator makes it easy to implement new features with few programming knowledge. This leads also to a very low time to market value where production-ready business applications can be deployed within days not months.

Platform as a Service (PaaS)

Salesforce provides both, Software as a Service and Platform as a Service.

Salesforce CRM is the world's first Platform providing SaaS whereby Force.com is a Platform as a Service(PaaS) product.⁷

Salesforce.com is providing various other platforms to customize Salesforce applications where the No-Code Platform is not sufficient enough. These platforms will be used for developing applications and websites which can then be easily integrated

⁶<https://trailhead.salesforce.com>

⁷Manchar and Chouhan, 2017.

into the main application. One of these platforms is the Force.com platform which focuses on creating applications with the proprietary programming language Apex, whereby another worth mentioning platform is the Heroku platform, which enables application creation in other well-known programming languages.⁸⁹

Force.com

Force.com is a platform provided by Salesforce.com. Salesforce uses the Force.com Platform to store all information securely in the cloud and it is designed to simplify the development and deployment of cloud-based applications and websites. All information is organized in objects and records, where an object is a tab on a spreadsheet and a record is like a single row of data. This platform has several features for the use of Salesforce.com's facilities. It provides developers the possibility to create multi-tenant add-ons for the main Salesforce.com application.

Since the Force.com platform is provided as a service, it has the advantage of requiring less maintenance for the developer and does not need manually installed software updates. Force.com applications are built using the proprietary programming language Apex.

Heroku

As briefly mentioned before, Heroku is a cloud platform that supports multiple programming languages.¹⁰ Heroku is part of the Salesforce Platform and supports Java, Node.js, Scala, Python, PHP, Clojure, Ruby, and Go where Ruby was the first language supported by Heroku.¹¹

2.1.2. Salesforce Programming Technologies

As shortly mentioned in the previous section Salesforce uses their in-house programming language Apex. Apex is one of the 3 core programmatic technologies from Salesforce, next to the Lightning Component Framework and Visualforce.

Apex

Apex is a strictly typed, object-oriented programming language with a syntax strongly based on Java. Apex enables the execution of flow and transaction control statements on the Lightning platform server in conjunction with calls to the Lightning platform API. Apex code is stored in the format of classes and triggers and is used for server-side development. These include button clicks, related record updates, and Visualforce pages. Apex Code can be initiated by web service requests and by object triggers.¹² Apex provides built-in support for common Lightning Platform idioms, including data manipulation and inline Salesforce Object Search Language (SOSL), which is

⁸Patel and Chouhan, 2016.

⁹Yin, 2019.

¹⁰<https://www.heroku.com/>

¹¹Patel and Chouhan, 2016.

¹²Patel and Chouhan, 2016.

similar to SQL and Salesforce Object Query Language (SOQL) queries that return lists of records.

Lightning Component Framework

The Lightning Component Framework is a framework similar to AngularJs and React for User Interface (UI) development. It is specially designed for developing dynamic Salesforce web apps for mobile and desktop devices. On the server-side, the framework uses Apex, and on the client-side Javascript.

Visualforce

With the help of the VisualForce Framework, it is possible to develop user interfaces that can be hosted natively on the lightning platform. The tag-based markup language from the framework is similar to HTML where each tag corresponds to a user interface component, such as a section of a page, record edit forms, or an input field. The behavior of these components can be controlled by the standard controllers or by extending the standard controllers with Apex.

2.2. Click-Based Development on Salesforce

In addition to Apex and the Lightning web components (LWC), click-based developments such as flows can also be used to develop Salesforce implementations. The Salesforce platform offers no-code and low-code features that simplifies the generation of customizations to applications. The Flow Builder and the Process Builder are two examples of these features. flows and processes can create, update and delete records. In case customer requirements are mostly aligned with the Salesforce standard, flows or processes can be used for implementation. Flows can be executed in different ways, depending on who the flow is intended for. Internal users, external users, or systems can execute a flow. Salesforce distinguishes 5 types of flows. These types are Screenflows, Scheduled-Triggered flows, Auto launched flows, Record Triggered flows, and Platform Event-Triggered flows. Screen flows require user input and have UI elements.

To deploy a flow as active, the flow must have 75% flow test coverage. This limit of test coverage does not apply if a flow is deployed as inactive. Nevertheless, Flow test coverage requirements do not apply to flows that have screens.

2.3. Definition of different Roles and their Responsibilities

In this work often the three terms Customer, (Salesforce) Administrator, and Developer are used. The next section will explain these three terms in more detail.

Customer

The term customer is often used in this work to refer to an individual or company which needs maintenance or new features in their Salesforce organization. Customers work closely together with the Administrator.

Salesforce Administrator

The role of a Salesforce Administrator is to work with the customer to define the requirements and customize and personalize the Salesforce platform accordingly. They are also responsible for the maintenance and upkeep of the platform. Furthermore, they should implement these requirements as simple and understandable as possible for users, no matter from which technical level. The administrator is also accountable for training the users. In this work, the term Consultant is used exchangeable with Salesforce Administrator.

Developer

A Salesforce developer works very closely with the administrator to implement the customer's requirements and meet their needs. A Salesforce developer is a developer who has a basic understanding of how Salesforce works to implement new features for the platform. As mentioned before there are several platforms, but this thesis is only focused on development using the Force.com platform where new features are developed with Apex, Visualforce, or Lightning Components.

2.4. What are Salesforce specifics?

As previously explained, Salesforce is a cloud computing provider. To get a slight overview of common CRM systems Figure 2.2 shows a few CRM systems grouped by on-premises and on-demand systems.

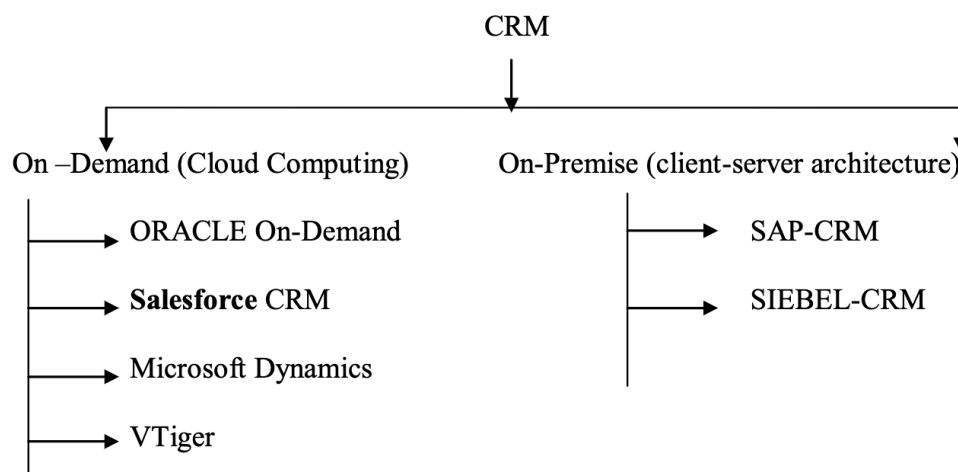


Figure 2.2.: Common CRM Systems illustrated distinguish by their architecture.

Source: Pullarao and Thirupathirao (2013)

Next to other well-known CRM systems, like Microsoft Dynamics, in Figure 2.2, it is shown that Salesforce also uses Cloud Computing. Salesforce Lightning Platform is a PaaS system that allows writing custom server-side or client-side code with the limitations that it can only run on Salesforce also custom defined database tables and fields can not be loaded into any other database than Salesforce. This means that tools for managing and updating other IaaS and PaaS systems cannot be used to customize Salesforce.

Before 2017 the only way to deploy changes was the org-based development where a Salesforce org itself was the source of truth.

The deployments were made with changesets. Salesforce provided a click-based User Interface (UI) to select, add and deploy components. There is also the possibility to show the dependencies of your already added components. Deployments can be made from any environment to another if they have set the permission for accepting inbound changesets. With this method, the same components can be deployed to multiple systems without the need for any local software installation on your local machine. The changeset deployment is not ideal for large deployments, because it becomes very complex and cumbersome to select the right components. The dependency list does not work properly within large deployments. In case of a failing deployment, for example, because of a missing component, it is not possible to just add the missing component and try it again. The whole changeset needs to be copied and then the missing component can be added and tried again. This is time-consuming and exhausting. Apart from this, a disadvantage is, that there is no possibility of version control or rollback options. Once a changeset was deployed, there is no way to reset the system to its original state.

Salesforce allowed its developers and administrators to specialize in creating business value and placed little emphasis on DevOps practices. In addition, the complexity of components and requirements continuously grows, resulting in more human resources and an increase in different environments. This can quickly lead to chaos in managing and synchronizing the environments and the deployment of features.

A huge difference from other platforms was the integration of version control. With version control, it is possible to track differences in file versions and have a change history of them. It also provides an extremely reliable backup under the assumption that it is used correctly.

The very first successful version control system was introduced in 1972 and was called Source Code Control System (SCCS) whereby Git was first developed in 2005.¹³

Nevertheless, Salesforce started the integration of version control for the first time in 2017, when the Salesforce Developer Experience (DX) was introduced. Salesforce DX has two main concepts. On the one hand, there is now the Version Control with also the possibility to create temporary development environments the so-called scratch orgs, which are explained in Section 3.1 and on the other hand the subdivision of code and metadata into packages. The Salesforce DX also includes new developer tools and

¹³Just et al., 2016.

extensions, especially for Visual Studio Code and Github. These enhancements made it possible to work with integrated development environments and command-line interfaces, which should make it a lot easier for development teams to work together.

Managing the Salesforce development lifecycle requires a unique skillset - the combination of the knowledge of a developer, an administrator, and an operator is needed. There are various well-established tools for managing and updating data but none of them can be directly used with Salesforce.¹⁴ Customized Salesforce Implementations often have a mixture of configuration and coding. These configuration settings are done by the administrator, which does not have experience with version control in most cases. This can make it challenging for the developer to synchronize the different environments. For this reason, the introduction of version control and Salesforce can be quite demanding.

Stack overflow, a public community-based platform of coding and technical-related question and answers, carries out an annual study to examine all aspects of the developer experience from career satisfaction and job search to education and opinions on open-source software. *"The challenge of managing Salesforce deployments was one factor that led Salesforce tied with SharePoint to the top of the list of "most dreaded platforms to develop on" in the annual developer survey from Stack Overflow in 2017."*¹⁵ One year later in 2018, Salesforce was still under the top three of this list.¹⁶

To summarise, this chapter was about Salesforce in general, the architecture with the three core concepts of Salesforce explained which are multi-tenancy, metadata, and APIs. The different Salesforce services and platforms with the different programming technologies were also mentioned. The last section dealt with the specifics of Salesforce and the difficulty to manage different environments.

To get an overview of the [Different Environments](#) provided by Salesforce the next chapter deals with these environments.

¹⁴Davis, 2019.

¹⁵Davis, 2019.

¹⁶<https://insights.stackoverflow.com/survey/2018technology>

3. Different Environments

When dealing with Salesforce for the first time, the different usage of terms like instance, organization or environment can quickly confuse. To understand and explain these different terms, the underlying architecture of Salesforce must first be understood. Salesforce is based on the concept of multi-client capability, which is explained in more detail in the previous chapter [Architecture](#).

Salesforce Instance

A Salesforce instance is a cluster of infrastructure which contains everything that is required to run Salesforce. A unit includes an application server, a database server, the database itself, and a file system. Salesforce groups customers together in so-called PODs. A Point of Deployment (POD) is also an instance of Salesforce whereby each customer is allocated to only one POD where their data resides. This means that a POD can hold multiple customers but a customer can only belong to one POD. Each geographic area has its instance. Currently, these areas are divided into North America, Europe, and Asia Pacific. There is even a separate instance only for sandboxes. The terms "instance" and "POD" are often used equivalent.

Salesforce Organization (Org)

A unique version of Salesforce that contains customer data and customized configuration is called a Salesforce org or organization. Each org can be highly customized and has its unique ID.

Salesforce Environment

A Salesforce environment is a Salesforce org for a specific purpose. Every customer will have at least one environment which serves as their Live System also called the Production System. Customers usually have additional environments like a development environment, a testing environment, and a training environment. These environments are also organizations but normally non production environments are sandbox instances. The different types of sandboxes are explained in more detail in the next section.

3.1. Sandbox Environments

"A sandbox is a copy of your organization in a separate environment that you can use for a variety of purposes, such as testing and training. Sandboxes are completely isolated from your Salesforce production organization. The operations you perform in your sandboxes don't affect

your Salesforce production organization. You can create different sandbox environments for your org, depending on your needs for storage, copy configuration, and frequency of refresh.”¹

Sandboxes are a copy of the production system but are a completely independent and isolated system. If changes are made by a developer or administrator to a sandbox, they have no effect on the production system. The same counts for data. If data is added to a Sandbox it has no effect on the production system but vice versa the data created on the production system can have an impact on specific types of sandboxes because some types are also cloning data.

In Salesforce there are 4 different types of sandboxes. Each type has a specific interval at which the sandbox is automatically updated. Furthermore, they differ in their properties regarding which data, classes, users, etc. are included. Another important aspect is the data size of the sandbox and the available licenses per sandbox type.

For development and testing in an isolated environment, a **Developer Sandbox** is used. This type of sandbox includes a copy of the production configuration but not the data. The metadata of the developer sandbox gets refreshed daily. This type of sandbox is for development purposes and unit testing. The data and size storage is very limited in this sandbox and for this reason, it is not suitable for regression or system integration testing.

A **Developer Pro Sandbox** is also for development and testing in an isolated environment but can store a larger amount of data sets. This sandbox can be used for integration and user testing as it has a storage size of about 1GB. The developer pro sandbox gets refreshed daily and also here is just the metadata synchronized.

The next type is the **Partial Copy Sandbox**. It is intended as a testing environment. The Partial Copy Sandbox includes the production configuration and a fragment of the production data. Which data the sandbox includes can be defined by a template. Quality assurance tests are commonly performed on a partial copy because besides that this type has 5GB of storage for files and data it also includes a subset of the production data. The refresh interval is 5 days.

Performance testing, load testing, and staging are only supported in **Full Sandboxes**. It can also be used for user acceptance testing, performance testing, and data migration tests. In addition, it is even used for user training. A Full Sandbox is a clone of the production environment. It includes all configurations and all data. This type also includes field tracking history. Every 29 days the full sandbox gets updated and gets in synchronization with the production environment.

The last different sandbox type is the full sandbox. It can be used for user acceptance testing, performance testing, and data migration tests. In addition, it is even used for user training. Every 29 days the full sandbox becomes updated and is synchronized with the production environment.

¹[Sandbox Licenses and Storage Limits by Type n.d.](#)

Depending on the Salesforce edition, the sandboxes are available in different numbers. In Table 3.1 the relation between the Salesforce edition and the number of available sandboxes per type is shown.

SANDBOX TYPE	PROFESSIONAL EDITION	ENTERPRISE EDITION	UNLIMITED EDITION	PERFORMANCE EDITION
Developer	10	25	100	100
Developer Pro	0	0	5	5
Partial Copy	Not available	1	1	1
Full	Not available	0	1	1

Table 3.1.: Available Sandboxes per Salesforce Edition
Source: *Sandbox Licenses and Storage Limits by Type* (n.d.)

Depending on the Salesforce Edition, the use of the sandboxes and the associated development process may vary slightly. Of course, the Unlimited or the Performance Edition would be preferable, as all sandbox types are available. Therefore an optimal development process can be run through, and consequently, a higher quality of the software tests can be ensured.²

However, it cannot be assumed that this edition is always available. Therefore, Chapter 9 specializes in the environment structure, taking into account that only a small number of different types of sandboxes are available.

Scratch-Org

In addition to the various sandbox types, there are also the **Scratch-Orgs**. The scratch org is a source-driven and disposable deployment of Salesforce code and metadata. Different Salesforce editions can be simulated with the corresponding functions, as Scratch-Orgs can be configured individually.

Since Scratch-Orgs are temporary and expire by default after 7 days, this can increase productivity and this expiration process ensures that teams regularly synchronize their changes with their version control system and work with the latest version of their project. Scratch-Orgs can also be used to start automated testing in combination with Github actions.

²*Sandbox Licenses and Storage Limits by Type* n.d.

4. Software Quality

The term software quality is defined by IEEE¹ as followed:

"In the context of software engineering, software quality is the degree to which a system, component, or process meets specified requirements, as well as customer or user, needs or expectations."

As Davis (2019) mentioned in his book, software quality can be divided into three areas. Namely, functional quality, structured or non-functional quality, and process quality.

The motivation to increase software quality can have many reasons. However, the two most common are cost and risk minimization.

Cost Minimization

The cost of detecting and fixing problems in software increases exponentially with time in the software development life-cycle which is graphically represented in Figure 4.1. Cost refers to time and resources in terms of human resources and money.

"Most defects end up costing more than it would have cost to prevent them. Defects are expensive when they occur, both the direct costs of fixing the defects and the indirect costs because of damaged relationships, lost business, and lost development time."

Beck (2000)

Compared to automated tests finding a failure during the manual quality assurance process, is time-consuming and expensive. Nevertheless, it is a lot cheaper to fix the issue in QA than if the end-user notices it while the system is already running which can increase the costs 30 times. The cheapest and most convenient time to find a bug is during the specification phase. A bug is a fault or error that occurs in the behavior of the software. Failures found during the implementation phase are easier for the developer to fix because the code is still fresh in their mind and they don't have to re-read it. This also makes it easier to solve more complex problems.

In addition to this, software with high quality is easier to understand, requires less maintenance, and can respond more cost-effectively to change requests.

Risk Management

Risk management is defined in the ISO 31000² as the identification, evaluation, and

¹IEEE Standard Glossary of Software Engineering Terminology 1990.

²ISO 31000:2018(en) Risk management - Guidelines 2018.

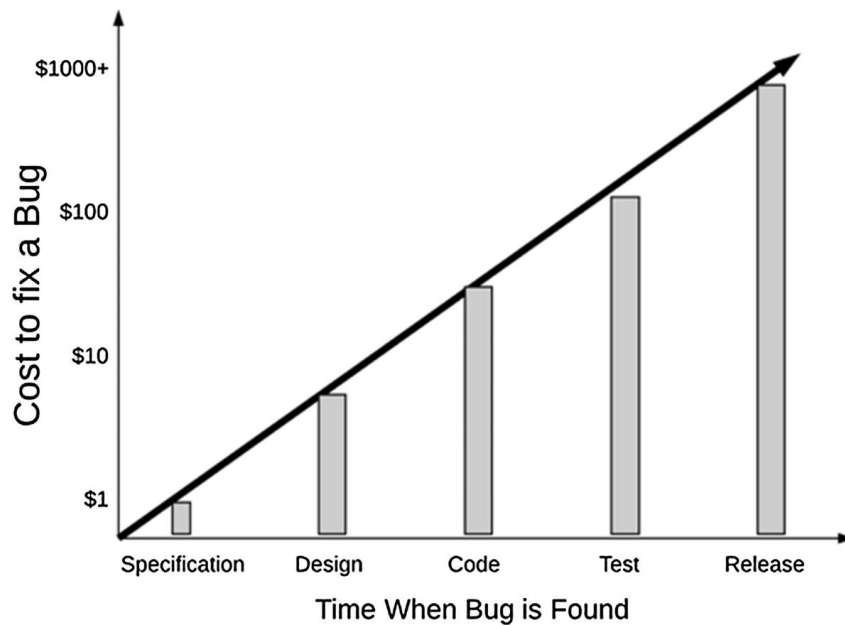


Figure 4.1.: Correlation between a late found error and costs.

Source: Davis (2019)

prioritization of risks. Although most software bugs are merely annoying and inconvenient, some bugs can still contribute to significant financial problems or even threats to humanity. To get an overview of common software bugs with significant consequences, Garfinkel (2005) wrote an article about the history's worst software bugs. The list is still growing. Software with high quality is also very likely to be less prone to errors and therefore leads to a lower risk.

4.1. Functional Quality

Based on functional requirements or specifications, functional quality reflects the extent to which the software complies with its particular design. In other words, does the software do what it is designed to do? In contrast to this, there is the **structural** quality which deals with the quality of the underlying implementation.

4.2. Structural Quality

The Consortium for IT Software Quality (CISQ)³ has defined five major structural characteristics to provide a business value to a software system. These characteristics are reliability, security, performance, maintainability, and size which are discussed in

³SOFTWARE QUALITY STANDARDS – ISO 5055 2021.

more detail below. Most of these aspects of structural quality can be evaluated only statically through the analysis of the software's inner structure and its source code.

Reliability is the probability that a system will operate correctly for a specified period. No repair may be required and performed during this time. Reliability follows an exponential failure law. It decreases with the amount of time considered for the reliability calculation. This means that the reliability of a system is high at the beginning and decreases gradually to its lowest value over time.

The **security** aspect is considered next. Software systems can be attacked to steal personal data, introduce vulnerabilities, monitor content, damage the behavior of software or suspend them completely (denial of service). Many security weaknesses originate from inappropriate architectural patterns and poor programming, these include SQL injection or cross-site scripting.

Another important aspect that is unfortunately often neglected is the **maintainability** of a system. Software developed under time pressure is often very hard to maintain. In retrospect, it is often very difficult to make small adjustments in little time and with as little effort as possible without fear of introducing new bugs to the system. This can be traced back to little to no documentation, no uniform naming, and not respecting certain programming patterns.

The **size** of the code also has a great impact on the quality and the complexity of the software. This includes the whole source code including database structures, scripts, and configuration files. A database architecture may fit a small application, but if it becomes larger over time, it can become completely unusable and unsuitable. Nevertheless, the size of the functionality has a huge impact, as the size of the functionality of the application increases also the size of the application's surface increases which means that there are more possible vulnerabilities in consideration of security.

Software is developed for specific use cases and users. Often, however, no consideration is given to how this software behaves in terms of memory, **efficiency** and speed if the amount of users or the functionality of the system increases.

4.3. Process Quality

Process Quality is concerned with how software is built, as opposed to Functional and Structured Quality, which is concerned about meeting the specified requirements.

To measure the effectiveness of a software delivery process the following four key performance indicators (KPIs) have emerged over time⁴:

- **Lead time:** Time to deliver completed work to end-users

⁴Davis, 2019.

- **Deployment frequency:** How often you update production
- **Change fail percentage:** How often updates negatively impact end-users
- **Mean Time to restore:** How quickly you can recover from a failure

These metrics are in line with the dual goals of DevOps to be fast and not introduce new errors, where the first two metrics are more related to innovation and fast releases and the second two are related to the reduction of bugs and stability of the system.

Lead time indicates how long it takes to deliver completed work to the end-user. The shorter the lead time, the faster feedback can be provided and possible errors found and can be corrected. Lead time can be considered as the sum of the time required to develop a new function and the time required to deliver that function to the end-user. This means a good development and deployment process has a small lead time.

The **deployment frequency** indicates how often configurations or coders are deployed to the productive system. This is directly related to the size of the deployment packages. For example, if a deployment only takes place every two months, the packages will be much larger than if every week a new deployment gets released. Very large packages also increase the probability that such a deployment will lead to errors. It could be hard to indicate which of the changes were responsible for the error to occur. These kinds of deployments are very tedious and painful. It must also be considered that not every change is of equal value to the customer, some are very valuable while others are almost insignificant.

*"Large batch size implies that valuable features are waiting in line with all the other changes, thus delaying the delivery of value and benefit."*⁵ A high deployment frequency can make deployments less painful and more pleasant as the package size can be minimized significantly.

With the **Change Fail Percentage** it is measured how often deployment errors occur in production that requires immediate remediation. Errors in the production system usually lead to stressful situations and also usually have a financial impact. To ensure that most bugs are sorted out before deployment, the underlying testing process must be very sophisticated. A very high change fail percentage can be an indication that the testing process needs to be adapted and extended.

Mean Time to Restore (MTTR), indicates how long a productive system is paralyzed after an error occurs. This time is related to the lead time. If there is a higher lead time, it is also possible to release a hotfix more quickly and thus reduce the MTTR. This metric is indirectly related to a good post-release bug system. If a bug occurs but is not properly recorded, it takes longer to start processing the bug. If the time between the detection of the failure and the signal to the developer is shortened, the overall MTTR is automatically shortened as well.

⁵Davis, 2019.

This thesis will address the three different types of quality to varying degrees. The first part of this work concerns analyzing the structural quality characteristics of the company which is discussed in chapter [Software Complexity](#). In this chapter, the characteristics of Code Size and Maintainability are explained in more detail.

The topics of process and functional quality are concerned in chapter [Gap Analysis of the current development process](#) where an analysis of the current process is described. This serves as a starting point for the process suggested in chapter [Deployment Strategy](#).

The last part of this thesis deals with increasing the functional quality of Salesforce implementations by choosing suitable testing methods and indicating at which phase of the development process they have the most impact. This can be found in the chapter [Conclusion](#). Trying out different testing methods and examining their success rate can be read in chapter [Empirical Experiment comparing different Software Testing Methods](#).

5. Software Complexity

Software complexity is a way to describe a specific set of characteristics of a code.¹ Throughout the entire life cycle of the software, development, and maintenance are associated with high costs. If the complexity of the software is reduced, the cost of the software will also be reduced.²

This chapter is about measuring the complexity of 3 different Salesforce projects. To do this the Cyclomatic Complexity (CC) and Lines of Code (LoC) are used which are described in detail below.

5.1. Code Complexity Metrics

Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through the source code of a program, which is computed by using the control flow graph of the code. This graph consists of nodes and edges where a node corresponds to groups of commands and edges are connecting these nodes.³

The Cyclomatic Complexity can be calculated as:

$$CC = E - N + 2$$

E is the number of edges and
N is the number of nodes.

CC can determine the upper and lower bound of test cases for branch coverage and path coverage. To achieve complete branch coverage of a method CC is the upper bound of test cases needed. Most of the time, the real amount of test cases is lower than CC because some paths may be impossible to reach. In contrast to this, CC can also be seen as the lower bound of test case numbers for path coverage assuming that each test case takes one path.

“The method normally uses McCabe cyclomatic complexity to determine the number of linearly independent paths and then generates test cases for each path thus obtained. Basis path testing guarantees complete branch coverage.” Westfall (2016)

¹Samli, Aydın, and Yücel, 2020.

²Yu and Zhou, 2010.

³McCabe, 1976.

To illustrate the cyclomatic complexity (CC) metric, the work from Graylin et al. (2009) "Cyclic "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship" is used as base. CC and LoC are used to analyze the code complexity of 3 customized Salesforce implementations.

The number of lines of code, in this case, is counted only from the apex code without code comments. The cyclomatic complexity was also calculated only from the Apex classes of a project.

Nevertheless, the complexity of [Click-Based Development on Salesforce](#) can also be calculated with the cyclomatic complexity. To keep in mind, flows and processes can also call sub-flows and other Apex classes. The complexity of these called objects will not be taken into account for the flow's overall complexity but can be calculated separately and summed up later on.

5.2. Static Code Analysis

Static code analysis is an analysis of a program without being executed, as opposed to dynamic analysis, which is performed during the execution of the program.⁴

For the scope of this thesis, 3 different Salesforce projects were analyzed. The analysis was performed using the PMD static source code analyzer.⁵ The PMD was chosen because it has already a standard rule set for apex applications available and it is free of charge and the results can be exported as comma-separated values (CSV) files. More about the static code analyzer can be read in section [Useful Tools to calculate Cyclomatic Complexity](#).

From the exported file only the lines where the total and the highest CC were displayed were extracted. After this step was completed also the number of Lines of Code per Apex Class was calculated and exported to another CSV file. In the next step, these two files were merged with the Apex Class name as the key. This resulted in the following diagrams from Figure 5.1 where the correlation between LoC and CC is shown.

In contrast to the study by Graylin et al. (2009) where it was mentioned that LoC and CC have a stable practically perfect linear relationship, it can be seen in the diagrams, that there is no direct correlation between LoC and CC. In the diagram of the third project, a high number of LoCs can be seen very frequently, but the corresponding CC is very low in relation. Whereby however in the first project the number of LoC is very high and also the CC is higher.

⁴Gomes et al., 2009.

⁵https://pmd.github.io/latest/pmd_rules_apex.html

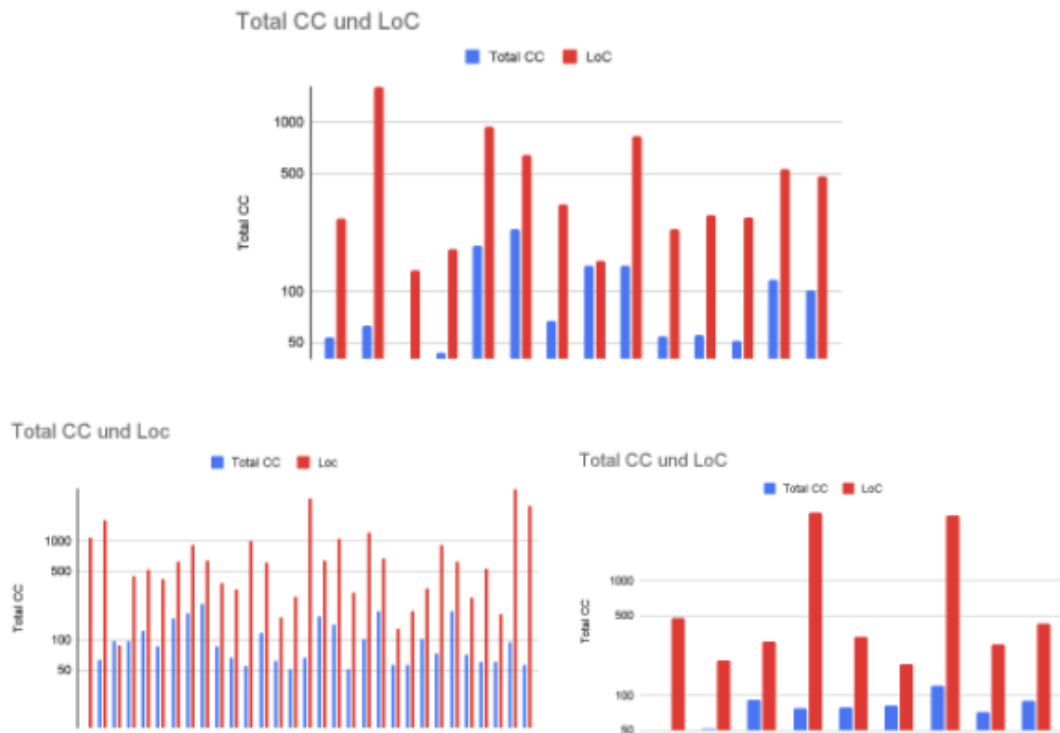


Figure 5.1.: Correlation between LoC and CC in 3 Projects

Thereafter, the focus was on the frequency of complex classes. For this, the data generated in the first step was reused. The highest CC of a class was taken and was classified into 4 categories namely Structured, Complex, Very Complex, and Not testable. The exact values can be found in the consecutive table:

Min	Max	Result
1	10	Structured and well written code
11	20	Complex Code
21	50	Very complex Code
> 50		Almost not testable

The first category means that the code is well written and structured. It has high testability and the cost and effort of testing are less. The second category has more complex code with medium testability and the cost and effort are also medium efforts. The effort and the costs are very high for testing very complex code like in category 3. The testability of the code is low. The last category is almost not testable and needs very high cost and effort for testing.

The categorization was done for every class in the project. In the following Figure 5.2 the above-mentioned classification is shown concerning the frequency of their

occurrence. This means if "Structured" has a frequency of one, only one class of the whole project has well-structured code.

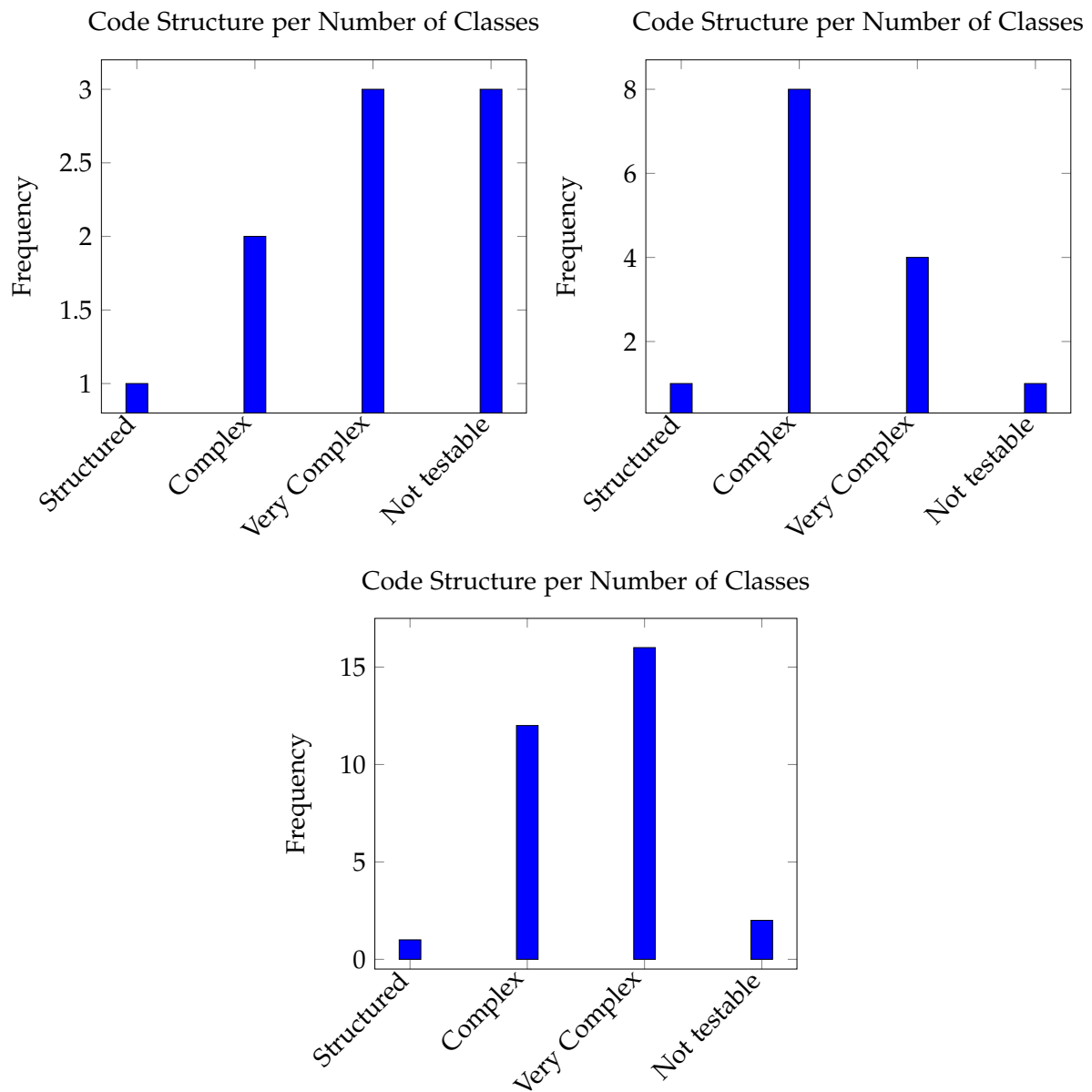


Figure 5.2.: Left: Project 1. Right: Project 2 Bottom: Project 3

It can be seen in the diagrams that the frequency of structured classes is very low in all 3 projects. For each of the projects which were examined the average CC per class is "Very Complex". This can be explained by the fact that until now low value has been placed on the quality of the code and therefore no quality assurance process is behind it.

5.2.1. Useful Tools to calculate Cyclomatic Complexity

To calculate the CC the Static Source Code Analyzer from PMD ⁶ can be used. This tool finds unused variables, empty catch blocks, and unnecessary object creation. Besides Java and Apex as the main languages used, PMD supports six other programming languages.

Directly integrated into the build process, PMD is most effective and powerful. Therefore also integration into the development process is considered. More about this can be read in the Section [Adaptions to the proposed Deployment Strategy](#).

5.3. Limitations of Salesforce

Since Apex runs in a multi-tenant environment, it is important to ensure that shared resources are not monopolized. For this purpose, the Apex runtime engine strictly enforces limits. If one of these limits is exceeded, a runtime exception is thrown that cannot be handled. These limits can be found on the official developer site from Salesforce - Execution Governors and Limits. ⁷

These Limits should be kept in mind, especially during performance testing. Performance testing is the evaluation of a system's performance in terms of responsiveness and stability under a given workload.⁸ Nevertheless, it should not be forgotten during normal functional testing what happens when several records are updated, deleted, or inserted at the same time.

⁶https://pmd.github.io/latest/pmd_rules_apex.html

⁷https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm

⁸Cui, 2021.

6. Software Testing

There are many definitions of software testing but it can shortly be defined as:

“Software testing is a technique for evaluating product quality and also for indirectly improving it, by identifying defects and problems.”¹

This chapter will be about the definition and the goal of software testing and the explanation of different testing types.

6.1. Definition and the Goal of Software Testing

Software testing is an important component of software quality assurance and therefore many organizations are spending up to 40% of their resources on testing. The percentage of this is even much higher for safety-critical software for example heart-lung machines or nuclear reactor control systems. A failure of those systems can lead to death or serious injury to people, loss, severe damage to equipment, or environmental harm.

Even with a normal software, an error that is not found can lead to many problems. The later in the development process the error is found, the more it costs in time and resources to correct the error.²

Figure 4.1 in Chapter [Software Quality](#) shows the impact of finding an error in terms of the cost of late fault detection.

The optimal and the cheapest time to detect a fault is to find and eliminate it during the specification phase.

Properly tested software products ensure reliability, security, and high performance which further results in time savings, cost-effectiveness, and customer satisfaction. Various test methods are applied to achieve this which differ in the type of tests and the time of application.

The next section will explain the classification of different testing methods.

¹Jovanović, 2006.

²Rubey, Dana, and Biche, 1975.

6.2. Software Testing Methods

In general, tests can be distinguished based on whether they test the behavior or the characteristics of a system if they are performed manual or automated, and on the interaction with the system.

In software testing, similar to software quality, a distinction can be made between functional and non-functional testing.

Functional testing evaluates the functional requirements of the product, including all functions, user interfaces, APIs, and error conditions.

Similar to the aspects of non-functional quality, non-functional testing also emphasizes the non-functional aspects such as efficiency, security, maintainability, recovery, and usability of the software.

To say it simply: functional testing is about what while non-functional testing is about how.

6.2.1. Functional Tests

When performing functional tests, test engineers verify that the functionality of the software conforms to the requirements specification.

To test functions, one of the most applied testing methods - black box testing is used. Black box tests specify inputs that are compared against an expected output without knowing the internal structure of an application. More about Black and White-Box Testing and their differences can be read in Section 6.2.3.

The advantages of functional tests include low granularity and little time required. Furthermore, this method can also be used for programs with a large codebase, since direct access to the code is not necessary. Over clearly defined roles it succeeds to separate the developer view and user view. An additional advantage of this method is that a large number of testers can use this testing approach without having background knowledge about the underlying implementation or knowledge of the programming language. The creation of functional tests can be started as soon as the specifications are provided. This type of testing also helps to detect errors and ambiguities in specifications at a very early stage.³

As it is shown in Figure 4.1 it becomes more and more expensive to find a failure in a later stage. The specification of the requirement serves as the basis for functional

³Bhat and Quadri, 2015.

testing. For this reason, it must be formulated very well and completely.

*"Without clear specification test cases are not efficiently designed and thus it all comes down to unambiguous specification analysis."*⁴

Functional tests are not suitable for testing the internal structure of the code, such as algorithm tests or branch coverage. To test these aspects there is non-functional testing.

6.2.2. Non Functional Testing

Non-functional testing, similar to non-functional quality, pays attention to non-functional parameters such as performance, usability and reliability. This type of testing is used to test the readiness of a system concerning non-functional parameters that are never considered in functional testing.

Both types of testing can be treated equally important as they both increase user satisfaction, reduce the production risk and associated cost of the software. Further non-functional test requirements are important and need a precise specification. Such a specification for a performance test would be, for example, the specification of how long the maximum loading time of an application is while it is being used by a defined number of users simultaneously.

The following table gives a small overview of different test types regarding functional and non-functional testing 6.1.

Functional Testing	Non-Functional Testing
Acceptance testing	Usability testing
Regression testing	Scalability testing
UI testing	Performance testing
API testing	Load testing
Integration testing	Security testing
Smoke testing	Stress testing
Localization testing	Maintainability testing

Table 6.1.: Types of Testing

Source: <https://rubygarage.org/blog/how-to-improve-your-product-quality-with-functional-software-testing>

Table 6.1 shows the division of the different test methods into functional and non-functional tests. The list of enumerated test types is not complete. It should only give a small insight into the most common types. In reality, there are over 100 different

⁴Bhat and Quadri, 2015.

types. On the following pages the most commonly used types are described in more detail.

Since the main part of this master thesis is focused on functional testing in combination with manual testing the functional testing methods are discussed in particular.

To complete the argumentation automatic testing will also be a short part of this thesis in combination with continuous integration and continuous delivery in Chapter 9.

Salesforce offers a built-in tool that can run tests. These tests can only be written in Apex. It is the general, the term Salesforce testing is used as synonym for testing apex classes and their triggers. Nevertheless, it must be considered as well that code-based testing does not only concern Apex but also Java Script which is implemented in Lightning components and Visual Force pages. More about the Salesforce technologies can be read in [Salesforce Programming Technologies](#). However, it is necessary to consider that there are a variety of testing methods when it comes to Salesforce testing like API and UI testing. The next 4 types are highly recommended for testing Salesforce applications and should be realized and prioritized even by smaller companies.

Acceptance Testing

Acceptance testing often called user acceptance testing, is the testing performed to determine whether the requirements of a specification have been met.

This type of testing is performed by the end-user on a separate instance as similar as possible to the live system with production-like data.

Usually, user acceptance tests are started only after the successful completion of a unit, system, and integration tests. Often it happens that all previously performed tests are successfully passed but the user acceptance testing is not successful. This can result from requirements being misunderstood by the developer or changes to requirements not being communicated clearly enough.

Regression Testing

With the help of regression testing, it is possible to find out whether the new functionality may affect existing functionality.

In regression testing, all or part of the tests that have already been executed is executed again to check whether they still work the same way as before. This testing is done to make sure that new code changes do not have side effects on the existing functionalities. It ensures that the old code still works once the latest code changes are deployed. Running all the test cases of a test suite can be very time-consuming. That is why there are different types of regression tests.

“Retest All” is a method for regression testing where all tests in the existing test suite are to be re-executed. Potentially, this requires a large number of resources and time. Another option is to select and prioritize the regression tests to reduce the regression test suite. Prioritization can be done in several ways and studies have shown that prioritization by code coverage, feature coverage, and previously failed tests is very effective.⁵

⁵Elbaum, Malishevsky, and Rothermel, 2000.

API Testing

An API (Application Programming Interface) is an interface that allows data to be exchanged between two independent systems. Therefore, it is important to perform API tests as soon as there is an interaction with other systems. Salesforce implementations often interact with other CRM or Enterprise resource planning (ERP) systems such as SAP. Another common connection is to an external ticketing system like Jira.

API Testing checks the API for functionality, reliability, performance, and security. Unlike UI tests, API tests do not focus on the look and feel of an application but on the functionality and correctness of the data.

API calls can be made directly from the code or special tools to validate the returned values can be used. Often the endpoints of integrated systems of the test systems and the live system differ because also test versions of the integrated systems are used. Of course, this must also be taken into account in the API test. Among the most common errors found during API testing are security issues, multi-threading issues, manhandled exceptions, unused flags or parameters, and wrongly structured return values in JSON or XML.

UI Testing

UI is the graphical user interface of a program or application. UI testing checks whether the user interface meets all the user's requirements. The most important aspects that are checked in UI testing include, Visual design, Functionality, Usability, Performance, and Conformance. Most often, UI testing is performed during manual QA and acceptance testing. However, it is also possible to automate UI testing. Some example tools which are very often used are Selenium, Cypress.io, and Puppeteer.

All above-mentioned Testing types are primarily Black-Box testing types and are used for functional testing. For non-functional testing, the combination of Black and White-Box testing can be used. Whereby the first method is mainly used by quality engineers without internal knowledge of the code and the structure of the application and the second method is mostly by developers themselves. The differences and an explanation of each method of White and Black-Box testing are explained in the next section.

6.2.3. Black & White-Box Testing

Black-Box Testing is also known as behavioral, opaque-box, closed-box, specification-based, or eye-to-eye testing. The main focus of Black Box Testing is on the functionality of the system as a whole. In **Black-Box testing** the tester has no knowledge about the internal structure or the source code behind the program, it is only based on the output requirements. "The tester is aware of what the software is supposed to do but

is not aware of how it does it.”⁶

Figure 6.1 represents Black-Box Testing.

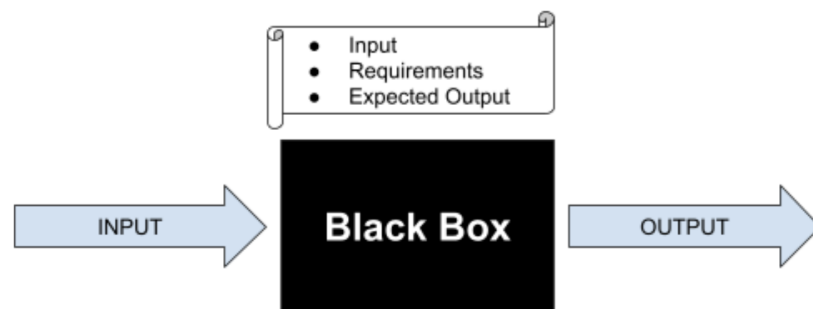


Figure 6.1.: Black-Box Testing explained

Source: Own Representation based on <https://wasistderunterschied.com/15-unterschied-zwischen-black-box-tests-und-white-box-tests/>

The first step of Black-Box testing is to examine the specifications and requirements of the System under Test (SUT). The test cases are then built around these specifications but first, some successful and some failing inputs are created. Then the tester determines the expected outputs to the given inputs. After these, the test cases are constructed around the specifications to check if the system operates accordingly with the given inputs. The real outputs of the system are compared to the expected outputs of the requirements.

Unlike Black-Box testing, in **white-box testing** the tester knows the internal flow of the program. White-Box testing is a method of testing the application at the source code level. Here, the test cases are specifically written so that all branches and functions of the code are called at least once. There are various methods to achieve this, such as branch testing, data flow testing, statement and decision coverage. In Figure 6.2 internal structure of the program is recognizable in contrast to Figure 6.1 where only the whole program is shown.

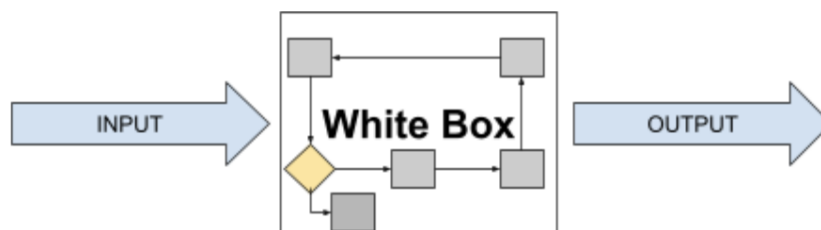


Figure 6.2.: White Box Testing explained

Source: Own Representation based on <https://wasistderunterschied.com/15-unterschied-zwischen-black-box-tests-und-white-box-tests/>

⁶Nidhra and Dondeti, 2012.

Comparison of Black and White-Box Testing

The following section shows a direct comparison between Black-Box Testing and White-Box Testing regarding required information, testing types, and when to use which technique.

Black-Box testing differs from white box testing in several ways. In Black-Box testing, the tester has no insight into the architecture or the underlying code, which is not necessary with this method. The goal is to check the behavior and functionality. Testing is done by a suitable Quality Assurance Engineer who has different testing techniques at his or her disposal. Black-Box testing methods are methods to perform system and acceptance tests on a higher level. Test reports contain errors in features and performance problems.

In contrast, there is the White-Box testing. Here the testers have full access to the code and the architecture. With this method, not only the behavior is tested, but also the logic behind it. Mostly this method is done by the developers themselves or other developers because they have knowledge of the programming language and the code. To be able to execute some methods, such as Basic Path Testing, the tester must have access to the entire code. Whereas in Black-Box testing external and functional tests are written, White-Box testing focuses on internal and structured tests on lower levels like unit and integration testing.

To get an overview of the currently used testing methods and the development process an analysis of the current and the desired situation of the mentioned company is conducted in the next chapter.

7. Gap Analysis of the current development process

A gap analysis determines the difference between an organization's current state of the business system and the desired state.¹ If the requirements of the desired state have not been fully met by existing policies, processes, or procedures of the current state, a gap exists.

Such an analysis can be performed by answering the following questions:

1. What is the current state of the system?
2. What is the desired state?
3. What is the gap between these two states and how can the gap be closed?

The table 7.1 gives an overview of the gap analysis from the company which was mentioned in section 1.1. This gap analysis was the starting point of what should be changed to improve the software quality of the company. The current situation of the different categories is explained in more detail below.

Version Control

Since there was standardized usage for sandboxes, there was also no standardized way of version control and no defined branching strategy. For every project, there were different branching strategies which makes it hard to switch from one to another project. No suitable backup strategy had been defined either.

Environment Structure

In some projects, only the productive system and a partial sandbox were used. On the partial, everyone worked at the same time. It may be that two developers were developing two independent features while the administrator was working on a flow while the customer was testing another feature. Other projects had at least a sandbox for each feature which was deployed to the partial when they were ready. On the partial then the customer was testing the features. As it can be seen very clearly there was also no standardized use of sandboxes but it was moving in the direction of feature sandboxes.

The positive aspect of this minimal sandbox structure is that the lead time to changes can be very shortened compared to more complex structures. Lead Time for changes is the amount of time it takes a commit to get into production.

¹*Quality management systems — Fundamentals and vocabulary, Requirements 2015.*

	Current Situation	Desired Situation	Actions to close Gap
Version Control	no uniform branching model is used	One well defined branching model applicable for every project	Definition of a branching model
Environment Structure	Different for every project	Standard basic structure which can be extended for every project	Definition of a basic standard environment structure
Testing Strategy	Random Testing is the most common Testing method	Use of various testing methods at different states of the application lifecycle	Analysis of different methods and definition of scope of application
Quality Model	Traditional Quality Model	Shift Left Model	Earlier emphasis on quality
Continuous Integration	Not present	Automated after every pull request creation	Creation of automated Actions
Continuous Delivery	Not present	automated after every merged pull request	Creation of automated actions
Deployment Frequency	After feature is ready and accepted by the customer	Fixed time slots for deployments, for example every 2 weeks on Thursday	Creation of a deployment plan, create Sprints
Failure Reports	No uniform way of failure reports	One way to register failures	Implementation of standard failure reports
Code Coverage	Only taken into account before deployment to reach the 75% limit	After a piece of code is finished, before it gets deployed anywhere	Defined Quality Gates
Requirements Definition	Requirements often incomplete	Complete requirements at the beginning of the implementation phase	Focus on requirements engineering

Table 7.1.: Gap analysis of the company's development process

Testing Methods

After the analysis of the company's testing process for over two months, the finding was that there was no standardized testing process or method established. As is commonly known, testing is the last phase in the classic waterfall model. It gets complicated when a project is delayed - then the testing phase is shortened or even canceled and this was often the case for the company as well.

Although the company follows an agile approach, the testing phase becomes often shortened or canceled.

The most common test method was randomized testing, which was used for about 95% of test cases. However, this method is not very target-oriented, since most tasks and projects are very dependent on user input.

Since Salesforce has the restriction that only code can be deployed into the live system that has a code coverage of at least 75%, often only attention was paid to achieving this specified limit through test cases without paying attention to the quality of the test cases.

Furthermore, the observation of the test cases revealed that there were many test cases without real checks and assumptions, this was the case in 50% of the written test cases.

In the remaining cases often no edge cases or requirements were tested, although the result of the program strongly depends on the user input besides a sequential aspect.

In addition, defined workflows or processes were only tested manually by the developer after implementation via the graphical user interface. The problem for this could be that the developer often did not have a sufficient specification available and also did not have the knowledge behind the desired feature so that the developer could react to all possible user inputs.

In terms of flows and processes, there were no written test cases mandatory. Only the underlying apex class, if one was used, needs a test case. Although for some types of flows it would be very easy to write automated tests. Record triggered flows are very suitable for writing automated tests, whereas screen flows are much more difficult and require more effort to run these tests automatically.

Quality Model

It was noticed that the company emphasized software quality in its development process at a very late stage. This is also shown in Figure 7.1. There are two models which are paying attention to quality. The traditional quality model, which only really starts to pay attention to quality after the code has been implemented which the company is using, and the shift left model. In the Shift left model, attention to quality is paid from the very beginning, for example at the specification stage. These two models are shown in Figure 7.1.

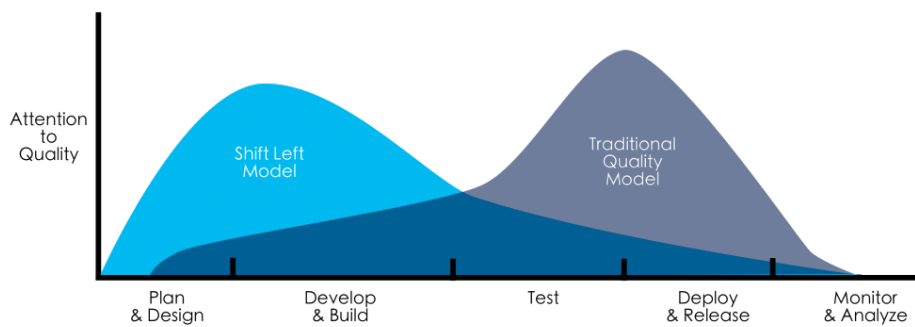


Figure 7.1.: Traditional quality model in contrast to the Shift left model

Source: <https://devopedia.org/images/article/121/8646.1538504518.png>

The goal was to move the company from the traditional model to the Shift Left model. To save effort, time and cost and to minimize the risk of late defect discovery, the shift-left method is perhaps the most reasonable.

Deployment Frequency

The company had a high deployment frequency on small changes but it could take very long for a completely new feature to be deployed into production. This could be because after the development of the feature no one felt responsible for testing the new feature properly. It was not clearly defined when an Administrator should test and when it is ready for the customer to test. Another point that led to delays was that there was no fixed deployment point where several features were deployed together. Each feature was deployed separately as soon as it was accepted by the customer.

Failure Reports

During the analysis phase, it was found that the company did not have a consistent post-release bug report. Some of the problems were sent by email, but there was no uniform recipient and the other part was entered directly via Salesforce. In other words, errors were either sent as an email to a collective email address or created as cases directly into Salesforce. This could be very difficult to react to a failure because no one felt responsible for it and this could also increase the number of time to restore service. This number measures how long it takes an organization to recover from a failure in production.

Code Coverage

Value was placed on code coverage only when a deployment into the productive system was pending since this is only possible with code coverage of at least 75%. In case multiple sandboxes were used, this means that it was possible to deploy code from one to another sandbox that had a code coverage of 0%. As mentioned before, it could take a long time for larger features to be approved by the administrator and ultimately by the customer. This made it even more difficult to write meaningful test cases after this long time to achieve the desired code coverage.

Requirements Definition

Requirements were often not fully defined before they reached the developer. It often happened that the developer had to ask for details or behavior in special cases, which led to delays. Also, the developer often did not have an understanding of how the new requirements will be integrated into the overall process and therefore could not create the correct test cases.

7.1. Proposals to close gap

To close the gap between the current and the desired situation a part of proposals are made in various chapters of this thesis. In Chapter 8, an empirical experiment to analyze, evaluate, and rank different software testing methods in terms of time, complexity, applicability, and errors found is conducted.

To this end, various test methods are examined and then evaluated qualitatively and quantitatively by experienced Salesforce programmers and classified accordingly. The focus is on black-box testing methods as there are many non Programmers in the company who can use these methods later on as well. However, administrators are also involved in the experiment to see the huge impact of poor or incomplete requirements definition. Administrators might become aware through the experiment how important it is to have a precisely formulated task specification. With a defined plan of what should be tested in which stage of the application life-cycle, the focus on quality will shift left automatically. The correct use of a variety of testing methods should lead to reducing or even completely closing the gaps.

To reduce the gaps in the categories of version control, environmental structure, continuous integration, and delivery proposals for a standard branching strategy and a definition of a basic standard environment structure are made in chapter [Deployment Strategy](#). However, continuous integration and continuous delivery are also part of this chapter.

Actions for the remaining categories are suggested in Chapter 10.

8. Empirical Experiment comparing different Software Testing Methods

The selection of a suitable software testing method compared to random testing leads to an increase in software quality, as possible errors can be found earlier, therefore costs and time can be saved. To evaluate this statement an empirical experiment is conducted. The outcome of this experiment is evaluated in the Section [8.3 Evaluation](#).

8.1. Goal

The aim of this experiment is to investigate on different software testing methods in terms of time, complexity, effort, number of test cases and errors found.

The aspect of time considered the time to write a test case and to learn the given test method. Possible previous knowledge and experience should also be taken into account.

Since custom Salesforce applications are very often dependent on user input and also often use click-based implementations such as flows and processes, the study is focused on Black-Box testing methods. Another reason for this decision is to be able to include people with little or no programming skills in this experiment since the company predominantly employs non-programmers and Salesforce offers a variation of no-code/low-code editors. The thought behind this is to relieve the programmers to some extent with testing so they can focus more on the quality of the programming tasks.

Therefore, 4 Black-Box methods are chosen which are analyzed in more detail. The participants of the study applied 4 methods but in overall the Random Testing is added as well. This means in total there were 5 different methods examined.

Afterwards it should be possible through the experiment to classify the test methods and to increase the software quality of the company by choosing a suitable test method for a particular use case.

8.2. Structure

In the following section the structure of the conducted experiment is described. First of all, the methods that were used is explained. This is followed by a characterization of the selection of subjects who participate in the study. Subsequently, the execution of the investigation is described in more detail. Following the description of the test methods used, the preparation steps are described in depth. Finally, the results of the study is described in terms of qualitative and quantitative results in addition to code coverage. At the end of this chapter some suggestions are listed to implement the methods which were analyzed into the application lifecycle.

8.2.1. Method

The experiment were conducted in two phases and were similar for developers and administrators.

Everyone received two task specifications. Administrators manually executed and logged the tests afterwards in a specially developed tool. Whereas the developers could decide if they wrote automated tests where it was possible or also executed the tests manually.

Once all methods had been tested, everyone received a questionnaire to evaluate the previously tested methods. The used questionnaire can be found in Appendix A. In order not to falsify the result of how many errors had been found by previous methods and to obtain the result more precisely, the methods were given to a group of test users in random orders for application. Also the order of the task were different.

Since the quality of the tests also depend very much on the specifications, there were also be a few evaluation questions about them in the questionnaire.

To test the success rate of the methods, the implementation differs from the task specifications. The fact that the implementation contradicts the specification and how many errors had been implemented is not known to the test subjects in advance. Besides this, the testers did not have any information about the success or failure rate of the other participants.

The basis of the experiment was a Developer Salesforce instance on which a Screenflow and several Apex classes were implemented to test the provided tasks. The application to be tested was a minimal version of a car rental company, where it should be possible to enter rental information and to create new vehicles.

In order to gain a better understanding of the code base, a statistical evaluation of the lines of code with respect to the programming language used can be found in Appendix B. Some Screenshots of the application and the implemented Screenflow are also provided.

8.2.2. Selection of the Participants

The objectives of this study set certain prerequisites for the participants.

Thus, the ultimate focus was on subjects who filled a role that at least had experience with Salesforce implementations.

For the automation of the test cases only Salesforce developers were considered whereas for manual testing both Salesforce consultants and Developers were examined.

Most of the testers came from the company itself but also external subjects were invited to the experiment.

8.2.3. Conducting the Examination

4 Black-Box testing methods were chosen to analyze them in the context of customized Salesforce implementations. Before starting the experiment, each participant got a short explanation and an example of each selected method.

To be able to compare the methods effectively, all test cases had been written for the same section of code or parts of the application. In total there were 2 different tasks to solve.

After all methods have been probed, each participant received a short questionnaire regarding previous knowledge, time needed, difficulty and errors found. The questionnaires were then evaluated and the methods were ranked.

Since the company mainly uses random testing, this method was also considered and finally compared with all others. For a fair comparison, random testing being performed with the same number of test cases then the average number of test cases needed overall. The next section briefly explain the chosen methods and some of their known advantages and disadvantages.

As (Bhat and Quadri, 2015) mentioned in his research that the Boundary Value Analysis (BVA) together with the Equivalence Class Partitioning (EP) and Decision Table testing form the basis of functional testing methodology, the experiment also concentrated on two of these methods. First of all, the focus had been on Decision Tables. After that a closer look at the Boundary Value Analysis, Error Guessing and Transition State Diagrams were given. Finally the random testing method and their advantages and disadvantages was explained.

Decision Tables

Decision tables are a visual representation to determine which actions should be performed depending on certain conditions.

Table 8.1 shows a simple example of a car rental company who writes a Decision Table about the invoice creation to find out when to add a discount or a subscription fee.

Actions		Rules							
Same Pickup and Return		T	T	T	T	F	F	F	F
Subscription existing		T	T	F	F	T	T	F	F
Accept Subscription		T	F	T	F	T	F	T	F
Conditions									
10% Discount			✓	✓					
Add Subscription Fee				✓				✓	
Create Invoice			✓	✓			✓	✓	
Update Account				✓				✓	
Display Error Message		✓			✓	✓			✓

Table 8.1.: Example of a Decision Table - Invoice creation

To create a Decision Table the following steps needs to be performed:

1. List test inputs and conditions in the table under conditions.
2. Calculate possible combinations with 2^n where n is the number of conditions. The number of possible combinations increases exponentially.
3. Fill the table with all possible combinations.
4. For all combinations of values, determine the expected result or subsequent action.
5. Create at least one test case for each column. If the rules are binary, a single test for each combination is sufficient. If a condition is a range of values, the Edge Cases should also be covered.

The huge advantage of the Decision Table method is that it can help to identify all possible combinations of a condition and find possible gaps in the requirements. It can also help to convert complex business rules into simple Decision Tables which can help the tester, the user and the developer.

However, the disadvantages must not be ignored. The complexity of the condition tables increases exponentially with the number of input parameters.

Boundary Value Analysis

Boundary Value Analysis (BVA) is one of the most common techniques in Black-Box Testing Techniques however it is only suitable for applications where the input is within a certain range. In this method the values of the boundaries of an input domain are tested.

This method checks both valid and invalid inputs. There is a very simple way for determining the limits, which is shown below.

A test case is written for each edge value. Boundary values are defined as follows:

- Min - 1
- Min
- Min +1
- Max -1
- Max
- Max +1

In some explanations of this method also a value in the middle of the range is used.

(Min,MAX) is the range specified for a field validation. The only failing test cases should be the one with Min -1 and Max +1, all others have to succeed.

Error Guessing

Tests are created based on the tester's skills, intuition, and experience with similar programs. Ad-hoc tests are still the most widely used technique. There are no special rules for this technique and it can be used at any level of testing. (Beer and Ramler, 2008)

The success rate of this method depends mainly on the experience and skills of the testers.

The following is a list of some common errors¹:

- Null pointer exception
- Index out of Bounds
- Division by zero
- Synchronization Errors

¹Vipindeep and Jalote, 2005.

- Entering no values and pressing submit
- Entering values in the wrong format
- Upload Files with wrong extension
- Upload files exceeding the max. or the min. limit
- invalid parameters

This technique should always be combined with other techniques to achieve better results. One of the biggest disadvantages of this technique is that the quality of the test case depends highly on the experience of the testers. Qualified and experienced testers are essential to perform these tests.

A major advantage of this testing technique is that it uncovers defects in areas that would otherwise go undetected by other formal testing techniques.

State Transition Testing

The state transition test is used to test a sequential input combination and the subsequent outputs. It is important to note that this is only possible for a finite number of input values.

A transition diagram usually consists of 4 parts. The States that the application can reach are represented by a square. The Transitions from one state to another are illustrated as an arrow. Events that trigger a transition are displayed as labels of an arrow. Actions that are the result of a transition are also represented by a square.

Figure 8.1 shows an adequate example of a State Transition Diagram. In this example, the simplified functionality of an invoice calculation of a car rental software is shown. If the user selects the same pickup and return station he gets a discount of 10%. It is only possible for already signed users to make a rental otherwise an subscription has to be charged beforehand. The booking can only be completed successfully if this fee is agreed otherwise the booking will be canceled and no invoice will be issued.

This diagram is also supported with color codes. Red indicates that something has been entered incorrectly or leads to a failing action.

To have it mentioned there is also the possibility to represent the state transition in a table, however this method is not taken into account in this experiment because it is very similar to the Decision Table and the focus was designing and presenting test cases as differently as possible.

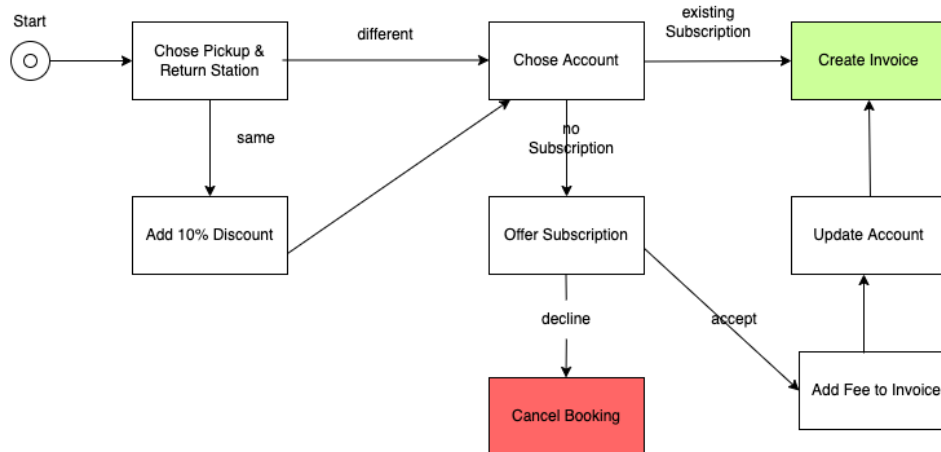


Figure 8.1.: Transition State Diagram of Invoice creation

Random Testing

As mentioned before, random testing is the main testing method in the company as it is performed when there is not enough time to write and execute other tests.

Inputs are randomly generated independently from the test domain. These inputs are then evaluated against the expected outputs.

Random Testing is really expensive with regard to test case number in comparison with Boundary Value Analysis. A study found out that for Boundary Value Analysis 8 testing results were sufficient whereby Random testing needed about 50 000 random test cases to equal the effectiveness of the Boundary Value Analysis. (Reid, 1997)

8.2.4. Preparation

In order not to disclose any customer-related data and to gave external persons the chance to participate in the experiment, a few preparations had to be made. To ensure the objectivity and reliability of the empirical experiment, care was taken to use only free and accessible software.

Salesforce Developer Org

A developer org for Salesforce is created and serves as a basis for the experiment. A developer org has all the features and licenses needed to get started with Salesforce. Anyone can set up a developer org for free. Also separate users with different permissions were created.

Github Repository

A git repository for the new developer org where every participant has its own branch to write test cases are created as well.

Automated Test Runs

To test and compare the selected methods as easily and straightforwardly as possible, a GitHub Action was implemented

The GitHub action reacted as soon as a new code commit was pushed to the repository. First, SFDX was set up on the Github server. After that, authentication was done from our Salesforce instance. For testing, a scratch-org was created which was then deleted again to release the resources. The code was pushed to the scratch-org. Afterwards the tests were automatically executed and evaluated.

Capturing manual Test Results

In order to record the results of the manually executed tests of the administrators and to make them traceable a tool for recording these test results is developed. Here the specifications were displayed and successfully or failed test results with their input and output parameters were added. However, if a test failed the reason why it failed or the expected output could be added. Later on, the customer should also have access to this tool to log test results, possible errors and missing requirements.

Prepare Task and Templates

A document with the task descriptions, the information the testers needed to use the application and a short introduction with examples of the chosen testing methods was prepared. Related links to the different methods were also added if the introduction was not enough.

To keep the effort as low as possible for the participants a templates to create Decision Tables and the Transition State Diagram was created.

Set up a questionnaire

The questionnaire was created with google Forms and serves as a basis for the evaluations of the different methods.

Part of the questions are evaluated with the 5-Point Likert Scale.(Preedy and Watson, 2010) The questionnaire template used can be found in Appendix B.

Perform Pilot Test

To make sure that everything works as it should, the tasks are understandable, the access permissions fit and the questionnaire works a pilot test was performed. Afterwards, this was evaluated and small things were adjusted to the setup.

8.2.5. Experimental Difficulties

As Reid, 1997 also mentioned in "An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing" such an experiment raises a few

questions. One of the most interesting question was the questions if the experimental test techniques were well-defined and repeatable.

For this reason it must be considered that well known and well defined test procedures were used and that only programs and tools that can be tested in a freely available application without additional costs were used. This means that everyone could reproduce this experiment with its own task ideas and methods. Another question that came up was if these test techniques are language-specific or can be applied to other languages as well. Reid did an similar experiment with different methods and base of his experiment was 20 000 lines of Ada code whereas the base of this experiment was Apex code. More about the results of the study from Reid (1997) in comparison with the results of this experiment can be read in Section 8.4.

8.3. Evaluation

This section presents the results of the empirical experiment from Chapter 8 where 4 Black-Box testing methods and their effectiveness had been evaluated in comparison to random testing.

In principle, it was about testing a minimal car rental booking system in Salesforce with a resulting invoice creation.

The experiment was divided into 2 tasks.

One task dealt with the maintenance and supply of the cars which were needed for creating a rental. Depending on the HP entered, an subscription class were then automatically selected that was later required for the calculation of the invoice which was used in the second task. Furthermore, it could be decided whether a car had a child seat or not and whether it is suitable for winter. The number of seats and color were further attributes that were rather secondary in the task specification.

The task of the participants in the experiment was to check whether this creation worked correctly and whether the correct subscription class was assigned.

This was implemented by a RecordTriggered Flow which triggered as soon as a new record of the type vehicle was created.

The second task was to check the booking and the invoice creation. A booking was made by selecting a customer, a start and an end date and a pickup and return station. Cars were only displayed if certain criteria were met, for example, in winter only winter-ready cars can be rented or cars with a high subscription class can only be rented to customers of a certain age.

Depending on the data entered, available cars were then displayed in a list. To complete the booking, one of these cars must be selected and then the invoice was created. The total price again depended on whether the customer was already in the system and had a subscription and whether a discount was approved. A complete description of the tasks can be found in Appendix B.

To realize this, a Screenflow was created. In this flow an Apex controller was implemented which queried the database of available and suitable vehicles. This means that the participants had to test 3 different types of Customized Salesforce implementations, namely Screenflow, RecordTriggerd Flow and an Apex Controller where only the components of the Screenflow were visible to them.

As presumed in the short explanation of the two tasks, some test methods were better suited for these tasks than others. The Boundary Value Analysis, for example, was better suited for the creation of the vehicles. However, it could also be applied to the booking. To provide an overview of how well the methods performed in terms of ease of use, time and errors found, some questions to the questionnaire were added.

The questionnaire was a mixture of qualitative and quantitative questions which were analyzed in detail below.

The first section of this chapter is about qualitative and quantitative analysis. The second part is a statistical analysis of the amount of test cases. The last part focuses on the quality of the test cases in terms of code coverage.

8.3.1. Qualitative & Quantitative Results

The general question analysis showed that 80% of the participants were developers. These 80% had a minimum of 3 years programming experience and a maximum of 13 years. This leads to an average programming experience of 6,5 years.

Whereas the 20% of Administrators have on average more than 4 years of experience with Salesforce the developers have only 2 years in average experience with Salesforce technology. One out of the developers stated that he has no prior experience with Salesforce.

Altogether 7 people completed the whole experiment including evaluating the two task, send back a list how many errors they found and also completed the questionnaire. Nevertheless there were still a few people who not completely finished every step but a part of it and also these results helped evaluating the methods.

Overall 90% would do such an experiment again and stated that they have learned new testing methods and will use this knowledge in the future.

Error Guessing

100% of the developers already had experienced the Error Guessing method and stated that they use this method regularly to test their implementations. On the other side administrators indicated that they never heard of this method. Probably, also the administrators know this method and they use them regularly but they just do not know the name of it.

Error Guessing was evaluated as the easiest to apply method. This was no surprise because there are no strict ruleset to apply. This method just relies on the experience

of the tester. With an average application duration of 20 minutes it was the second fastest method after the Boundary Value Analysis to apply.

80% of the participants agreed that this method is more effective with a low number of test cases than random testing. The result of the effectiveness can be seen in the Table 8.3 below.

It was also stated as a quick and easy to use method to find usual errors in a program. Furthermore, errors can be captured with this method that no other method would find was mentioned. The biggest disadvantage which was named was that this method relies so much on the experience of the tester and for new testers it can be really difficult to find errors.

Decision Table

The experiences with this method were very divided. One quarter never heard of this method, one quarter heard of it but never used it, one quarter used it once and the last quarter use it regularly. This also explains why this method has the highest average application time of about 28 minutes. The Decision Table was also stated as the most difficult method to apply but 75% of the participants also said that they will use it again. Nevertheless this method was also rated by 75% as more effective than random testing.

The most mentioned advantage was that this method gives a good overview of the task specification and lists all possible methods. It was also mentioned very often that this method was very complicated to understand the first time but it gets much easier if its done several times. According to the findings and the discussions afterwards the rating of the difficulty and the average time to apply would be very different if the experiment would be examined again with the same participants.

The participants figured out that this can be very time consuming with a larger amount of input parameters as the number of test cases increases exponentially. However it was declared that this method is a really good method for critical task to find faults but it is of course very time consuming.

State Transition Diagram

The next method was the least known method. Only 33% used it once and the rest 67% never heard of this method. The good thing was that even if the majority did not know this method the complexity rated not really high. Also the average time to apply this method was about 25 minutes which is not really high if it is considered that this was the first time people used it.

Opinions about the future use were divided into thirds. One third will definitely use it again, one third will use it again sometimes and the last third has a neutral opinion about it.

Even this method is not really known 66.5% strongly agree with the effectiveness over random testing.

A view participants mentioned that it is an excellent method to get an overview of the task specification but the first time it a bit difficult to construct such a diagram. For the second task it was much easier because the participants already knew this method.

Boundary Value Analyze

40% use the Boundary Value Analysis regularly, 20% used it once and the rest 40% have never heard of this method. As expected this method was really easy to understand. As mentioned before this method was the fastest to apply with an average of about 11 minutes.

The distribution of the probability to use this method again and the effectiveness in comparison to random testing were 60%to 40%. 60% strongly agreed with these statements and 40% only agreed with it.

With this method a high amount of errors were found but it is not applicable for every use case. Many errors in combination with the task specification were not found.

8.3.2. Statistical Analysis

Table 8.2 presents the number of generated test cases used for Error Guessing, Decision Table, Transition State Diagram, Boundary Value Analysis and Random Testing. These values are the mean values from the experiment over all participants. To have a fair comparison the number of test cases for random testing is calculated as the mean value over all numbers of test cases as stated in the formula below.

Task	Error Guessing	Decision Table	Transition State	BVA	Random
1	6	10	8	3	6
2	4	10	4	7	6

Table 8.2.: Average number of test cases per method

As there can not be half test cases the result was rounded to the next whole number. This approach of comparison is the same as in the study of Reid, 1997 with the difference that there every method was separately compared to random testing.

$$\#RandomTestcases = \frac{(\#DecisionTable + \#TransitionState + \#ErrorGuessing + \#BVA)}{4}$$

According to the fact, that the number of test cases can be at least calculated for the BVA and Decision Table, the number of test cases is predefined. However, the amount of test cases trough the participants varied a lot during the experiment. For the Decision Table there is a formula depending on the input parameters with 2^n where n is the number of input parameters and for the BVA we always have min-1, min,

min+1, max-1, max, max+1 and optionally one value from the middle. Nevertheless, there was actually a large fluctuation in the number of test cases per participant. This could be explained that some participants divided the task into 2 tables and others just combined or omitted parameters. Another explanation to this is that a part of the participants have tested all boundary cases and others only the outer boundaries. This explains the fluctuations of test case numbers with the BVA.

There were also strong fluctuations in the Error Guessing method depending on the experience level of the tester. It can not be clearly say if an experienced tester needs more or less cases because on the one hand the tester exactly knows which errors are likely to happen and this may decrease the number of test cases but on the other hand this could also lead to more test cases because he has already a lot of experience and therefore a lot of ideas how an error can be found.

As it is shown in Table 8.2 the Decision Table needed in both task the most amount of test cases, but with this method the most errors were found at least for task 1.

With the Decision Table and the Transition State Diagram, many missing specifications were found which could have been fixed early in the application life-cycle. This would not be possible with the other methods because they already need the finished implementation to create the test cases. According to the fact that the BVA was not really suitable for the first task, this is also reflected in the number of test cases. In order to show the effectiveness of an unsuitable method for a specific task, this method was nevertheless part of the experiment. As already expected, it was not very successful. However, the second task was a very good example for the application of this method and several errors could be found. Since this method can be very time-consuming when there are several subgroups. To address this issue different variations of the method were examined afterwards. More about this method can be read in Section 8.3.3. Task 2 was not really suitable for the Decision table and the Transition State diagram.

Table 8.3 shows the probability to detect all errors of each method grouped by the examined task. As already mentioned the BVA was not really suited for the first task. This is also represented in the table where BVA had the least probability of finding all errors. In comparison to this, BVA was the most effective method in the area of the second task whereas the Decision Table could not even find 20% of the errors. As it can be concluded, if the Decision Table method fits good for a task, the BVA will not really succeed in this special use case and the other way round.

Task	Error Guessing	Decision Table	Transition State	BVA	Random
1	48.98%	59.18%	22.45%	9.52%	28.57%
2	66.67%	16.67%	33.33%	72.22%	33.33%

Table 8.3.: Probabilities of error detection of the different methods in comparison

Figure 8.2 is another representation of how many errors with which method were found. In the graph it can be seen that the Decision Table leads to the best result with 4 errors out of 7 were found which leads to a detection probability of about 60%. In contrast to this the Decision Table method was the least successful in task 2 with about 17% error rate. For task 2 it seems that Error Guessing and the Boundary Value analysis are the best suited methods. However, it should be noted that Error Guessing is highly dependent on the experience of the testers. If less experienced people had participated in the experiment, the result might have been very different. One participant had a lot of experience in software development and software testing. However, the experiment was his first point of contact with Salesforce. Although the test cases for Error Guessing were very well thought out and defined, it was not possible for the developer to find any errors. In contrast to the Error Guessing method, the BVA should always provide equally good values, since there is an algorithm how to apply this method. Nevertheless, even with this method, there were different approaches. This resulted in a fluctuation of the errors found, but also fluctuations with regard to the time required but more about this deviations can be read in Section 8.3.3 where the quality of test cases is discussed in more detail. The result for random testing was very surprising as many errors were found at first but after a closer look the errors were all of the same category for example 3 from 8 test cases generated a car with HP less than 50 HP, which should not be possible. At the end this leads to just only one failing test case out of 8.

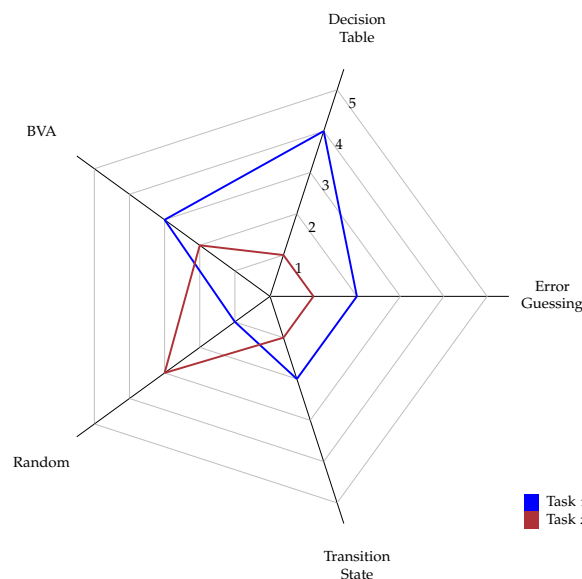


Figure 8.2.: Number of Errors per Method found

At the end of the survey every participants had to vote which method is the best for them and which is the worst. Everyone had to sort the 5 methods according to their priority with respect to usability, difficulty and errors found. Surprisingly Random Testing was always voted in last place. This result was quite unexpected, because Random Testing was always used as the main testing method until now.

Through the experiment it became clear to the participants that Random Testing is not very effective and with little effort as for example with the Boundary Value Analysis many errors can be found. In the following list the placement of the individual methods can be read off.

1. Decision Table
2. Boundary Value Analysis
3. Error Guessing
4. State Transition Diagram
5. Random Testing

It could be noticed that the Decision Table is very popular, although it was rated as difficult to implement. In the comments it was also mentioned that this method is very good to evaluate the specifications and to find out missing information. Nevertheless it was also stated that this method is good for smaller tasks because it can get very complex with more input parameters.

A part of the programmers even mentioned that they will create this table in the future before they start programming in order to ask for possible ambiguities right away and to complete them.

One point where everyone agreed, no matter if programmer, tester or administrator, and where 100% was achieved, was the priority of the specifications. It came out that a complete specification is very important for everyone and indispensable for an error-free implementation.

8.3.3. Analysis of Test Case Quality

In order to check the quality of the generated test cases, a closer look at the comparison between the individual methods and random testing in terms of code coverage was taken. It would also be very interesting and conclusive to analyze the test code quality in terms of mutation testing, unfortunately this is not included in thesis but more about Mutation Testing can be found in Section [10.5](#).

Since automated tests were not part of the experiment, only manually generated test results were handed in. However, with the provided input parameters it was possible to write test cases and automate them.

Since this would be very time consuming to do for every participant, a report from one participant was selected and used as the basis for the comparison. On the other hand it is also not possible to automate test cases for screen flows in Salesforce without UI Testing tools therefore the Screenflow was rewritten into a record triggered flow.

Since the results of random testing can vary greatly, the generated tests run 10 times and then calculated the mean value.

The implementation for the first task consisted of a trigger and an Apex controller. First of all, since there was a logical error in the implementation of the first task, one part of the code could never be reached and therefore the Apex Trigger could never reach 100% of code coverage. The unreachable code measures 10%. For task two there was also an Apex trigger but here it was possible to reach all 100% .

Method	Code Coverage		Errors	Number of Test Cases
	Trigger	Controller		
Random Testing	72.5%	92.5%	3	11
Decision Table	90%	100%	6	16
Error Guessing	67%	85%	4	5
Transition State	75%	85%	2	6
BVA	62%	80%	0	3

Table 8.4.: Code Coverage of Random testing, Decision Table, Error Guessing, Transition State Diagram and the BVA method

Table 8.4 shows the code coverage, found errors and the number of test cases of task one from one specific participant. This explains why the numbers of errors can differ in Table 8.4 where the result from only one participants was considered in contrast to Figure 8.2 where the average values were calculated. The Decision Table was divided in two tables with 3 input parameters each. Instead of 64 possible combinations if this would be done in one table there were only 16 test cases.

If a look at the table is given it seems that the results are quite good especially for the Error Guessing method only one test case did not find any errors and this leads to an error detection probability of about 80%. But this result treated with caution because there is no fixed algorithm behind and the number of test cases can differ a lot as mentioned before. Also the result for random tests looks very good if it is considered that it almost don't need any resources for this kind of testing. A closer look to the results, showed that all errors were of the same type and therefore only counted as one error. This realization reduced the probability of finding an error from 37% to 9%. The same analysis was applied to the Decision Table and it led to 37% percentage of error detection probability.

The errors could also be divided into different categories. For example they could be distinguished between Implementation Errors and Specification Errors. Half of the errors were specification errors which means there were wrong or no specification given. This type of errors could have been found and avoided very early in the development life cycle. It is the cheapest option if an error is already found during the specification phase. With this testing method this is possible and can help reduce cost and time.

The analyze for code quality of the Boundary Value Analysis was applied on the second task because this is the perfect operational area for this method. The results were very exclusionary and interesting. Three different types of Boundary Value Analysis were examined and every method got 100% code coverage, but found a different

amount of errors.

The normal Boundary Value Analysis was the first one to investigate. For this method 21 test cases were needed because a mean value was always tested as well. Of these 21 test cases, 2 resulted in errors and the code coverage was as mentioned 100%.

However, since this was a large number of test cases, the idea was to use a minimized type of BVA as some of the participants also did. In this method, only the outer boundaries were tested. This method finally led also to 100% coverage with only one third of the number of test cases compared to BVA but the downside was that only one error was found. This result was also not very satisfactory, therefore a third method was analyzed.

The third method was a combination of the Boundary Value analysis and Equivalence Partitioning. Equivalence Partitioning testing divides the input values into different partitions in such a way that all values in a partition can be treated similarly or equally. This means that, only one value from each partition needs to be tested and for the rest of the values in the same partition it can be assumed that they will be treated the same way by the application. If one value is tested and the test is positive, it can be concluded that all other values in the partition have the same result and do not need to be tested separately. This is also true for failed test cases. If one value triggers an error, we can assume that other values in that partition will also trigger this error. In this case, this led to 5 partitions if only EP was used. Two invalid and 3 valid partitions. With the EP method it does not matter which value of one partition is tested. It is widely known that boundary values often not behaving as desired. Therefore the EP was combined with the BVA where for every partition the upper and the lower value was tested. The result were 8 test cases with 100% code coverage and 2 errors were found. This result was the most satisfactory of the 3 methods

In the experiment some participants mentioned that they think that the BVA is a really good, fast and easy to use method but they did the BVA only with the outer boundary values because of time lack which could also be possible in practice. With this combination a very good and efficient result had been achieved and only needed one test case more compared to the minimized BVA where only one error was found.

The summarized result with the number of test cases, code coverage and errors can be found in Table 8.5.

Method	Code Coverage	Errors Found	Number of Test Cases
BVA	100%	2	21
BVA minimized	100%	1	7
BVA + EP	100%	2	8

Table 8.5.: Code Coverage of different types of BVA

8.4. Appropriate method depending on the Application Lifecycle Phase

This section deals with the definition of the best fitting testing method based on the current phase of the application lifecycle. First it is explained how the Software Development Lifecycle is defined and in which phases it can be divided and afterwards the methods which were examined during the empirical experiment will be suggested to the use in the different phases.

8.4.1. Application Lifecycle

According to ISO/IEC 12207² the Software development life cycle (SDLC) is a framework designing task performed at each step in the software development process. The typical SDLC consists of the following 5 phases which are shown in Figure 8.3 and explained in more detail afterwards.



Figure 8.3.: 5 Phases of the software development lifecycle

Source: <https://www.inflectra.com/SpiraTeam/Highlights/Understanding-ALM-Tools.aspx>

Plan

The planning phase includes all aspects of project and product management including resource allocation, costs, project scheduling, defining requirements. Project managers, customers and also developers are involved in the planning phase. At the end of the planning phase the requirements should be clear and requirements specification should be approved from the customer. Also the defining and designing of the requirements are part of this phase and must be finalized before the next phase begins.

²ISO/IEC/IEEE 12207:2017 Systems and software engineering — Software life cycle processes 2017.

Develop

After the requirements are clearly defined the actual writing of the program starts. Depending on the size of the project one more developer is working on the requirements. It is important that all developers follow the defined guidelines and practices of the company.

Test

In this phase it is checked if the code meets the defined requirements. This phase is usually a subset of all stages in the modern SDLC models.

Deploy

Once the testing is finished successfully the goal is to make the product available for the end user. Most organizations do not deploy directly to the production environment. They move the product through different environments like testing and staging environments. With these two additional environments it is possible to catch errors before releasing the product to the final customers.

Maintain

After the product is acknowledged and deployed to the production environment the maintenance phase begins.

Depending on the outgoing software development process, the SDLC can vary. On the whole, however, the different phases can be transferred to the respective models. There are several different software development life cycle models. For example the agile software development lifecycle model is a combination of iterative and incremental process models. The difference to the previously explained model is that it is possible at every phase to go back to the previous phase.

8.4.2. Relationship between evaluated Results and the Lifecycle Phase

The test methods are evaluated in the order of the result of their popularity.

Decision Table

As mentioned many times before, the use of a Decision Table can provide a very good overview of the completeness of the requirements. It would therefore make sense to start creating the Decision Table in the planning phase in order to identify missing specifications as early as possible. Another advantage would be that the developer does not have to think about the different input combinations when he is generating tests, this results in time saving and he can concentrate on the real implementation. The use of a Decision Table also makes sense in the testing phase, since here special emphasis is placed on functional tests.

Boundary Value Analysis

Depending on the specification, also this method can make sense to prepare already during the specification phase. This makes it easier for developers to address possible boundary cases more specifically in their implementation. When the development phase is complete, there is often no time to test all possible use cases. This is why the BVA method is often used in the development phase as well.

Error Guessing

The Error Guessing method is most effective when using during the development phase. It can be used in the testing phase as well but only in combination with other methods which are focusing more on the functionality of the code.

State Transition Diagram

The State Transition Diagram is very similar to the Decision Table and can be used in the same phases. In addition, this method shows possible use cases that were not considered in the Decision Table, therefore it makes sense to use it during the Acceptance Testing phase as well.

State Transition Diagrams can also be very helpful in developing regression tests to graphically show the interaction between different features.

Random Testing

Random tests can be performed during every phase, with exception to the planning phase, of the life cycle as they can be easily created and automated with little cost. Especially in the deployment phase stress tests can be performed with random data on the different test and staging environments.

9. Deployment Strategy

This chapter is about establishing the connection between the different environment types and version control. The first part describes what a Git workflow is and how it can be applied. After that, it is about defining a uniform usage of Salesforce environments and their types under the assumption that at least the Salesforce Enterprise Edition is available. Then, the synchronization of these two methods is discussed and shown in more detail concerning refresh and renew intervals of the environments and branches.

9.1. Git Workflow

Atlassian defined the term Git Workflow as followed:

“A Git workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner.”

Implementing version control and choosing a suitable branching model, can lead to different problems. Selecting a branching method means nothing more than defining how the team will work together with version control in the future. However, it should be noted that a wrongly selected strategy can reduce the productivity of the team and prevent the successful implementation of a version control process. Therefore, the explained aspects below should be considered before selecting a strategy.

Team size is one of the most important points to consider. Small teams tend to use a very simple form of branching strategy to implement changes quickly. This can increase the deployment frequency and at the same time decrease the lead time to changes.

An example of a very simple branching strategy would be the feature branch strategy where there is a separate branch for each feature. When the feature is completed, it is merged directly into the master branch with a pull request. With this branching strategy, it would be sufficient to have a production system and create a separate sandbox for each feature.

This approach would no longer work for larger teams, as there are usually different environments to manage and also several roles such as administrators, testers, developers, and quality assurance engineers.

The implemented feature must pass through several levels to be accepted by the

customer and finally deployed to the production org and the master branch. This also means that in this case several sandboxes must be used.

The next thing to consider is the **frequency of releases** in the company. For example, if the release is daily, it will be very difficult to follow a complex branching strategy and use different systems.

However, if the release is weekly or even monthly, it is easier to go through all the defined phases of the branching model.

Of course, the **complexity** of the selected model should not be forgotten. Basically, the simpler the better. More complex models require more maintenance and it is also more difficult to get new employees used to and trained in this strategy. Not only for new employees it is difficult to adapt their old easy workflow to a new more complex one.

This leads to the next aspect which must be taken into account when choosing a new strategy - the **current workflow**. If the old workflow only consists of the live system without any testing environments it is very difficult to establish a new workflow with these things. The question of maintenance will also arise, for example, who is responsible for the creation and update of a new environment.

There is not only one solution that fits for all but there are many suggestions that can be adapted to the team's needs.

The following strategies are not the simplest ones possible because there are many different roles in the company's hierarchy. It is assumed that the team works in sprints of 2 weeks and consists of at least two developers, one administrator, and one customer.

9.1.1. Branching Strategy

In this section, a modified version of Gitflow is explained. Gitflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches. In this version, the User Acceptance Testing (UAT) branch was introduced because often Features get approved internally but not externally from the customer and this would decrease the deployment frequency.

In this model, there is the main branch that contains the current version of the live system and stores the official release history. All other branches are created from this branch. When a Sprint starts, the Quality Assurance (QA) branch DEV_QA is created from the Main branch. This branch contains the complete history of the project regardless of whether a feature has already been accepted by the customer or not. This branch serves as a test branch for the administrator. Here all features come together which have already been tested and marked by the developers. To create a new feature, a feature branch is created starting from the DEV_QA branch. Normally only one developer works on this branch.

Once the feature has been completed from the developer's point of view, he can create a pull request to merge this feature into the DEV_QA branch and thus release it for testing by the administrator. This pull request must be reviewed and accepted by another developer before it can be automatically merged. However, if this request is rejected, the developer must implement the proposed changes and create a new request. This continues until the request is accepted. This kind of review can increase the code quality.

Once the feature has been accepted by both developers and has been successfully merged, it is now the administrator's task to test this feature. First individually and then depending on the already implemented features. If the feature is not accepted, because it brings problems in connection with other already successfully tested features, this is communicated to the developer. The developer merges the current version of the DEV_QA branch into his feature branch and starts working on this feature again.

If the feature is accepted by the administrator, it will be pushed to the UAT branch. All features that have already passed the internal test are now on this branch.

The next step is to inform the customer that a new feature is ready for testing. Only when the customer approves this feature for the current Sprint it will be pushed to the Release Branch. On the Release Branch are all features that have been accepted for the current Sprint and will be pushed altogether to the Main Branch at the end of the Sprint. However, if the feature is not accepted, it is possible to fix it in the same sprint. For this, the UAT branch is merged into the corresponding Feature Branch and everything starts from the beginning.

At the end of a Sprint, all completed Feature Branches are deleted. Now it depends on which stage the incomplete feature is in. At the beginning of a Sprint, all Branches are again created and updated starting from the Main Branch. If the feature is still under development or waiting for the review of the other developer, the feature branch remains and the new changes are merged into the branch. However, if the feature is already at the DEV_QA or UAT branch and waiting for approval, the corresponding branch is merged with the current changes. The feature branch is deleted. If the feature is not accepted, a new feature branch is created from the corresponding branch.

This procedure ensures that no features are lost but at the same time the latest version of the code is always used.

In the case of an occurring error, out of the main branch a Hotfix branch is created on which the error can be fixed. After this was tested by the developer, he creates a pull request which must be accepted by the administrator. By accepting the pull request, these changes are not only pushed directly to the main branch but also to the DEV_QA, UAT, and the release branch.

This described Branching strategy is visualized in Figure 9.1.

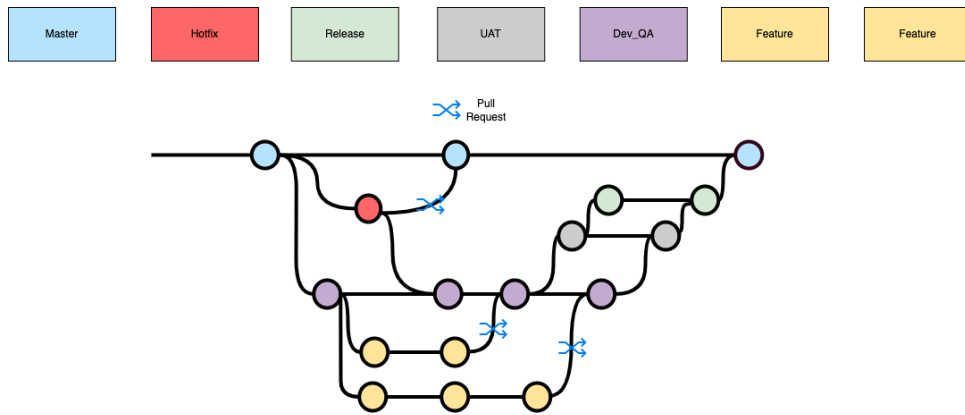


Figure 9.1.: GitFlow Branching Model

9.2. Environment Structure

Similar to the Git workflow, there should also be clearly defined rules and suggestions for the use of sandboxes to make collaboration as efficient and effective as possible. Since the number of different sandbox types depends very much on the Salesforce Edition used, the following diagrams are created with the assumption that at least an Enterprise Edition is available. With this edition, there is at least one partial copy available in addition to the normal developer sandboxes. This setup can be customized and extended as needed by the company but should provide a guide and basic setup for use.

The basic setup always consists of 5 different environments which are shown in Figure 9.2 and are very similar to the git branches.

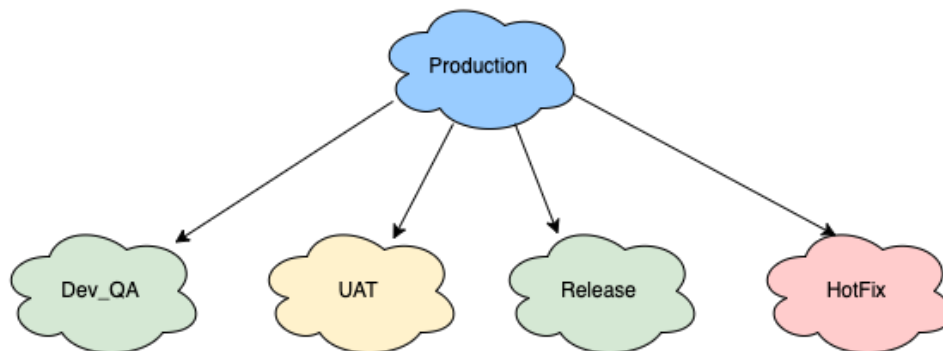


Figure 9.2.: Initial Organisations

Setup - Initialization:

For each project, the same structure should be created from the beginning. The production point is always the starting point. The different sandboxes are then created from this point. Except for the Hotfix sandbox, no changes may be made to the code in these sandboxes. Changes may only be made via deployments whereby it does not

matter whether a deployment is done with a changeset or the command line with an XML File.

The Dev_QA and the Release Sandbox are Developer Sandboxes. Since the customer ultimately determines whether a feature is accepted or not, the UAT is a partial sandbox since data from the live system can also be imported here.

The Release Sandbox is only used to release the accepted features from the customer. In the case that problem and an error have occurred in the live system, a Hotfix Sandbox is created for this purpose. Depending on the size of the bug, a scratch org can also be created.

After the setup is finished the system is ready for the first sprint. During the sprint, a distinction is made between the different types of requests. Generally, a distinction is made between feature request, change request, and Hotfix which are described in more detail below.

Feature Request:

After a new feature request is made by the customer, a new Developer Sandbox is created based on the current Dev_QA Sandbox. All changes concerning this feature are made in this feature sandbox. In concrete terms, this means that developers and administrators both work on the same sandbox. When the feature is ready and has been tested by the developer, it is deployed to the Dev_QA. On the Dev_QA it is then tested by the consultant individually and in connection with already existing features. After the feature was accepted, the feature package is then deployed to the next sandbox, the UAT. In this sandbox, all features are ready for the customer to test. After the customer accomplished the testing, the package is deployed to the release sandbox and remains there until the sprint is over and the deployment of all collected features to the live system is started. By using the Release Sandbox it can be ensured that only features that have been already approved by the customer are deployed to the live system.

If only the UAT would be used, one would always have to wait until all features have been accepted, and only then can be deployed. This could decrease the **deployment frequency** and also increases the **lead time for changes**. For example, if 4 features are released for testing on the UAT and 3 of them have already been accepted at the end of the Sprint and 1 has not yet been accepted, all 4 features would be moved to the deployment of the next Sprint, but this is not the case if a separate Release Sandbox is used.

Change request:

Change requests are small requests for already implemented features to change that need no longer than 2 days in total including testing and deploying to other Sandboxes. This definition can differ between companies. This procedure is very similar to that for a feature request. The only difference is that feature Scratch-Orgs are created instead of feature sandboxes. The rest of the process is the same.

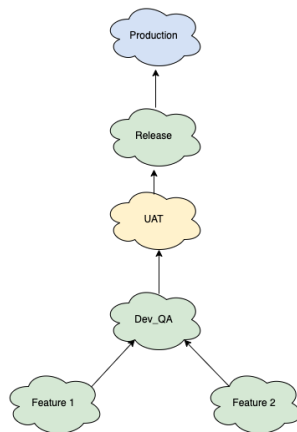


Figure 9.3.: Deployment cycle of a new feature

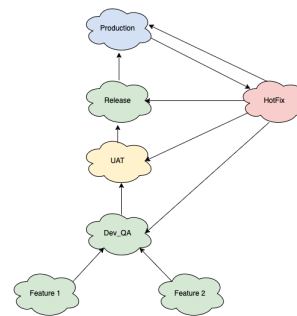


Figure 9.4.: Deployment cycle of a hotfix

Hotfix:

A Hotfix, as the name suggests should be fixed as soon as possible. A Hotfix should not affect the running release cycle and can therefore be deployed directly from and to the production system. Once the Hotfix is fixed it should also be deployed to the other systems like DEV_QA, UAT, and Release. Depending on the difficulty and size of the problem, a whole developer sandbox or just a scratch org can be created. The assessment of the problem is up to each developer.

The figure shows the comparison between a feature request and a Hotfix. On the left in Figure 9.3, it can be seen that the path runs from the bottom to the top, whereas on the right Figure 9.4, everything starts from the highest point, the productive system and is then distributed to all other systems.

9.2.1. Refresh Lifecycle & Clean Up

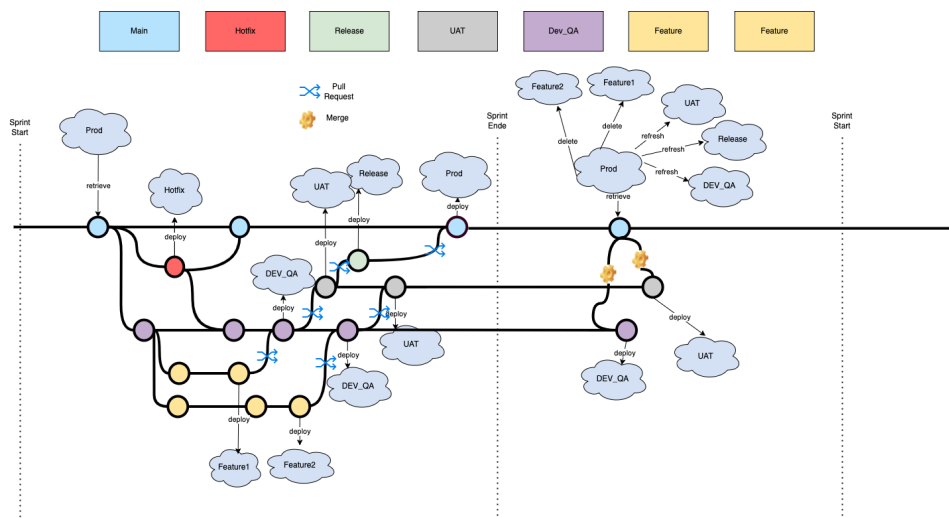
After a sprint is completed, all sandbox instances must be cleaned up and updated in addition to the branches.

Features that have been accepted by the customer and have already been deployed in the last sprint can be considered as completed and therefore the corresponding feature sandbox and the git branch can be deleted. In this case, the other sandboxes can be renewed or refreshed without hesitation. A refresh is mandatory because it is possible that some changes were made in the production organization from the customer like adjustments to the layout or renaming of labels. However, changes such as field validations or triggers must also be taken into account. These changes could prevent other features from working as expected and therefore synchronization and refresh are essential.

After a sprint ran without problems and all planned features were deployed, there is no problem with the refresh of the sandboxes. However, if a feature is not completed

for various reasons, the question also arises as to how this can be dealt with during the refresh.

9.2.2. Synchronization of Version Control and Environments



As mentioned in the previous section, all sandboxes are renewed or updated after each sprint. Before a developed feature can be merged to the DEV_QA Branch, it must first be validated and deployed to the feature sandbox. After that, a pull request can be created which automatically validates the deployment to the DEV_QA sandbox.

This procedure continues all the way up to the live system. To be able to deploy changes to the next higher environment or Git branch, a pull request must be created which triggers validation of the deployment and if accepted, deploys the changes

directly to the intended sandbox and branch. The techniques of continuous integration and continuous delivery can help to make this automatic synchronization possible.

9.2.3. Continuous Integration

Continuous Integration is a software practice in which changes to the code are uploaded to a central repository at regular intervals and simultaneously validated and automatic tests are executed. This method is intended to increase software quality by detecting and correcting errors at an early stage. Continuous integration could also help the team to deliver updates to their customers faster and more frequently.

The basic idea of continuous integration is to ensure that all changes made by the team have been tested and validated to ensure that a deployment-ready version has always been pushed to the current branch. This can be done by setting up a workflow that performs a validation deployment to a pre-authorized Salesforce organization.

This method can be implemented very easily using Github actions. The best time to validate the deployment is when a pull request is created. Depending on the outbound Branch, the organization on which the validation should take place is selected. For example, if a pull request is created from a feature branch, the validation of the feature to be deployed will take place on the DEV_QA sandbox. Only if the validation is successful, this pull request can be accepted and merged.

Continuous Integration with Github Actions on Salesforce

Figure 9.6 shows how a Github action for Continuous integration could look like on Salesforce.

1. The first step is to set up SFDX on the server. To use it, it must be downloaded with the **wget** command and a directory for it must be created with **mkdir**. After that, it can be installed.
2. The next step is to authenticate with the DevHub. To make this possible an access token is needed which can be created with the command **sfdx force:org:display -u <OrgAlias>**. The result is ideally saved in a file. There are also other authentication methods like using Github variables. No matter which way is chosen, afterward it is possible to authenticate as usual.
3. After successful authentication, a scratch org can be created.
4. This org does not contain any code. To change this it must be loaded into the org with the command **sfdx force:source:push -u <OrgAlias>**.
5. Once the code has been successfully loaded into the Scratch org, validation of the deployment can begin with the command **sfdx force:source:deploy -checkonly -testlevel "RunLocalTests" -json -p "Deployments/deployment.xml" -u <OrgAlias>**. The **-p** parameter specifies the path for the xml file containing the components to be deployed. If the **-checkonly** parameter is omitted, the deployment would

```

name: 'deployment validation'

on:
  pull_request:
    branches: [DEV_QA]
    types: [created]

jobs:
  build:

    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Write the token into a file
        run: 'echo force://PlatformCLI:*****@*****.my.salesforce.com > token.txt'

      - name: Install Salesforce CLI
        run: |
          wget https://developer.salesforce.com/media/salesforce-cli/sfdx-linux-amd64.tar.xz
          mkdir sfdx-cli
          tar xJf sfdx-linux-amd64.tar.xz -C sfdx-cli --strip-components 1
          ./sfdx-cli/install
      - name: Auth with the DevHub
        run: 'sfdx force:auth:sfdxurl:store -f token.txt -a DevHub -d'
      - name: Create scratch org
        run: 'sfdx force:org:create -f config/project-scratch-def.json -a DevHub'
      - name: Push source
        run: 'sfdx force:source:push -u DevHub'
      - name: Validate the deployment
        run: 'sfdx force:source:deploy --checkonly --testlevel "RunLocalTests" --json -p "Deployments/deployment.xml" -u DevHub'
      - name: Delete scratch org
        run: 'sfdx force:org:delete -u DevHub --noprompt'

```

Figure 9.6.: Example GithubAction for Deployment Validation

directly start without validating it first. Fail validation would cause the entire pipeline to fail.

6. Finally the Scratch org is deleted again to free up space.

After successful continuous integration, continuous deployment can now be introduced. Continuous deployment refers to the fact that any code that passes automated testing and validation is automatically deployed to the production system. However, there is a difference between continuous deployment and continuous delivery. Continuous deployment means that developers regularly push new code to quality assurance (QA) and operations teams for testing. Continuous deployment typically involves a production-like deployment area, and there is often some time between release and review, manual acceptance of changes, and release of new code to production.

If the code was then successfully merged, another Github action can ensure that this code is automatically deployed to the designated system.

This type of Github action only runs if the QA manually accepts the Pull Request therefore this is a continuous delivery method.

This automation ensures that the branches are synchronized with the sandboxes. Another advantage of this type of automation is the administrative effort. Since

administrators also create and accept pull requests, it is easier for them to deploy the change to the corresponding sandbox without touching the underlying source code.

Continuous Delivery with Github Actions on Salesforce

The Github Action for continuous delivery is built similar to Figure 9.6. The first difference is when the workflow will run. The first workflow will run every time a pull request is created on a specified branch - in this example branch DEV_QA. The workflow for continuous delivery will only run if a pull request was successfully merged. This means it is not sufficient to start the workflow only when a pull request has been closed, it must be additionally specialized that it triggers only when it has been successfully merged. How this can be achieved is shown in Figure 9.7 which was taken from an example of Github Docs ¹.

```
on:
  pull_request:
    types:
      - closed

jobs:
  if_merged:
    if: github.event.pull_request.merged == true
    runs-on: ubuntu-latest
    steps:
      - run: |
          echo The PR was merged
```

Figure 9.7.: Github Action on merge Pull Request
Source: Github Docs ¹

The next major difference is that in step 5 the `--checkonly` flag must be removed otherwise this workflow will also only validate the deployment but never actually deploy it.

These actions can be extended as desired. Another suggestion would be to automatically send an email to the responsible developer when an error occurs in the workflow. Only one example of an action on a particular branch was shown here. To use this action as described in the Gitflow model above, several queries would be needed to be implemented or one action per branch would have to be created to make sure everything gets deployed to the correct Sandbox or finally to Production.

¹https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#pull_request

9.2.4. Load Sample Data

Most of the features are very dependent on what data has been entered. The only sandbox in this model that also synchronizes data is the partial sandbox.

This means that no data for testing is available on the QA and feature sandboxes and would have to be recreated each time. This would lead to a lot of additional work since the sandboxes would have to be renewed or updated after each sprint.

Another aspect is data security. In the partial copy, the data is loaded directly from the production system. In the above model, the partial copy is the same as the UAT, only the customer has access to it for testing. Here there are no problems with data security. However, if all data were loaded into all other sandboxes as well, this could lead to privacy issues.

The Salesforce CLI provides some data-related commands to export and imports small amounts of data for development and testing to address this problem.

These commands are very easy to use but have the limit that only 200 records can be imported per command. With these commands, it is possible to export-related data in a single step without having to worry about the hierarchy of the data. So it is not necessary to explicitly recreate the links during the import. A JSON file is created which represents the data hierarchically without the use of ids.

Test data can then be generated on a sandbox and exported using the Data Tree Export Command. The exported files can then be placed on a separate git branch to be accessible to a developer in the future. Here is an example of what such a command could look like to export contacts and the associated accounts:

```
sfdx force:data:tree:export -p -q  
"Select Id, Name, (Select Id, LastName, FirstName FROM Contacts)  
FROM Account" -u <OrgAlias>
```

The parameter -q specifies whether it is a query or a file path. The parameter -p specifies that a separate file is created for each object, but also a separate plan definition file to facilitate the collective import. If no default organization has been set, the parameter -u must also be specified, which specifies the organization from which this data is to be exported.

These exported files can now be easily imported into any sandbox with the command:

```
sfdx force:data:tree:import
```

9.2.5. Deployment Frequency

The waterfall process results in an extremely long and costly feedback cycle. With such a long feedback loop, it is tedious for users to deal with existing bugs, and it is tedious for developers to remember and fix old bugs they wrote about almost a year ago. From an economic perspective, it becomes exponentially more expensive to fix bugs that have made it into production as shown in Figure 4.1.

Therefore release cadences should be as short as possible to reduce the time between an error being introduced and the error being found.

To reduce the chance an error can make it to production and get recent feedback practices like Paired Programming, Continuous Integration, and Test-Driven Development can significantly increase the chances of finding and fixing bugs earlier.

Figure 9.8 shows the correlation between costs and length of the feedback cycle.

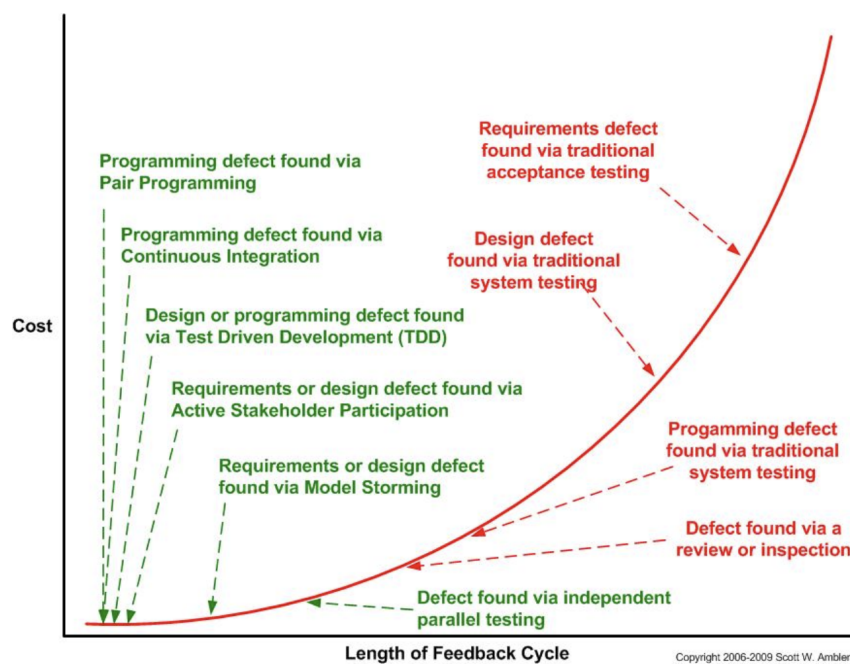


Figure 9.8.: Correlation between Cost and Length of Feedback Cycle

Source: Examining the Cost of the Agile Change Curve by Scott Ambler

As it can be seen, the cost is increasing exponentially with the length of the feedback cycle. With the Pair Programming method, one receives direct feedback, and thereby also very fast errors can be discovered and eliminated however, it must be considered that here additional resources are needed since per task always at least two developers are needed.

At the same time, continuous integration also gives the developer very fast feedback at an early stage, which can be reacted immediately. The introduction of short release cycles and the associated practices for obtaining rapid feedback can reduce the likelihood of bugs getting into the production system, but this in turn can create other problems. A part of these problems is briefly addressed in section 10.4.

9.2.6. Testing Strategies and Responsibilities

The administrator is responsible for ensuring that the specifications have been recorded by the customer in a complete and comprehensible manner. After this has been done, it would be useful if the customer reviews them again and adds additions, and correct them if necessary. Once the specifications have been finalized, the developer can start implementing them. At the same time, the administrator can prepare for test case creation. Depending on which task is to be processed, he can, for example, create a Decision Table or perform a Boundary Value Analysis. These methods also often make it clear that requirements are missing from the specifications. With this method, possible errors in the specifications can be detected and corrected very quickly. Section 8.4 can provide additional information on which method would be best suited for which task and on which phase of the application lifecycle.

9.2.7. Considerations

As mentioned before, the number of sandboxes depends very much on the edition used. Therefore, it must always be considered what alternatives there are for the used sandbox type if these are not available. It is important to consider the company structure and its customers. It is easier to use a better version of the sandbox than to downgrade the version.

Therefore, in the above model, only developer sandboxes and a partial sandbox are used. This model can be easily upgraded by, for example, creating a Full Copy Sandbox instead of the Developer Sandbox for UAT and using the Partial Copy as DEV_QA.

Another aspect that must be taken into account is that employees are already used to a certain workflow.

Therefore, it is important to communicate openly and honestly why the new workflow, which may involve more effort, will be used in the company in the future. It can also help to face the problems with the current workflow and state scenarios like what if something unexpected happens, is there a backup, or how to deal with errors in production.

However, it must also be made clear that the changeover cannot work from one day to the next and it will be a continuous process to achieve the desired results.

Even if the motivation to change is strong, the changes can lead to a feeling of overwhelm and anxiety due to a lack of knowledge of version control or sandbox structures. Which in turn can be seen as a major barrier to entry.

To address this issue, it is important to allow sufficient time for the transition and to make the transition as easy as possible with training and presentations. It can also help not to apply these changes directly to the whole company structure and to familiarize only one team with the new way of working for the time being. This allows other teams to see the benefits of these methods. Lessons learned from the team can also be shared, which can make the transition easier for the other teams.

Overly complex branching strategies can scare people off. Therefore, care should be taken to keep the branching strategy and the resulting sandbox structure as simple as possible but as complex as necessary.

These strategies should not be proposed by one person but should be discussed in a round with at least one person from each department that will be affected by the change. Other groups of people may have different views and objections which are not taken into account by a single person.

The last aspect that must be considered but which is important is that part of the configurations differs between the organizations such as integration endpoints. It must be defined how such special cases are handled to not overwrite these configurations on a refresh.

The mentioned company already started to implement a part of the above-suggested changes. The next chapter provides an overview of problems and possible solution approaches.

10. Adaptions to the proposed Deployment Strategy

The company had already started with some adaptations of their deployment process in regards to the proposed procedures with a focus on the increase of software quality. During the implementation, they had been confronted with some problems which are described in this chapter and for some of them, a solution is already provided. However, also, a few additions are suggested.

10.1. Synchronization between Version Control and Environments

Version Control serves as the basis for automation. Without version control, it is almost impossible to start test cases and quality gates automatically. Version Control also provides an extremely reliable backup which is another important aspect to use it.¹ Accordingly, it was important to define standardized handling of version control as a first step. Implementing Version Control takes some time and also brings obstacles. Therefore the company started with this change already with the start of the project which is explained in chapter [Background and Previous Work](#) at Section 1.1 of this thesis.

The company follows the Gitflow Branching strategy which is explained in Chapter 9. Depending on which Salesforce edition is available, it is decided which type of sandbox is used. The different types are explained in Chapter 3. Moreover, depending on the customer some additional changes were made but in principal every project has now the same base structure.

As mentioned in the earlier Section 2.4, Salesforce DX was first released in 2017. Before that, and still today, it is possible to modify the code directly through the Developer Console via the browser. It is also not possible to turn this feature off or temporarily lock it for some users. This complicates the use of Git, as it must always be considered to pull these manual changes from the organization beforehand.

However, the use of Version Control in the company already works very well and hardly causes any problems. Nevertheless, sometimes the temptation is too huge

¹Davis, 2019.

for some employees, and a part of the changes are made directly in the browser. This leads to the fact that the sandbox and Git are not properly synchronized. As a consequence, these changes could become overwritten if the developer not explicitly calls the command for retrieving the code directly from the org.

As a solution to this problem, a simple but very effective script was developed. The script performs exactly the above-described synchronization by first stashing the local changes of the developer. Then it pulls all the data which were in the developer's repository from the respective organization and commits it to his repository. To receive back the developer's local changes, they are inserted again with git stash pop. This can lead to merge conflicts which can be fixed directly in the editor. However, if in the meantime changes have been made by another developer which has been synchronized with Git but not yet deployed to the sandbox, the script throws an error indicating that the repository should be updated first.

10.2. Linting

Linting is a language-specific static analysis method that automatically identified possible structural faults in performance style or security which can be part of the CI process or can be applied in real-time directly in the code editor. This is a quick and inexpensive method that can run as often as needed to give the developer direct feedback on the code. Linting applies generic rules to the codebase.²

It is also possible to define a own ruleset or decide which ruleset or subset of a ruleset should be executed. If linting is applied directly in the editor feedback is provided immediately and the developer can respond and change the code accordingly and do not have to wait for the response of the CI/CD pipeline. This is a real good example of the shifting left method which was explained earlier in Figure 7.1.

Linting was used for Section 5.2 to analyze the cyclomatic complexity and the lines of code of the company's codebase. The results can be found in Figure 5.1 where the relation between LoC and CC is displayed and Figure 5.2 where the degree of the code structure is shown.

This modification of the development process was one of the easiest and least time-consuming ones because it needed just an installation of a plugin in the editor immediately results could be seen.

²Davis, 2019.

10.3. Administration of Bug Reports

Before starting with this master thesis, post-release bugs were announced by the customer directly via mail or the community portal. Thereupon a customer case was created in the internal Salesforce which was then processed. With the community portal, it is possible that the customers can directly create cases in the company's Salesforce system.

As mentioned in the previous chapter, some preliminary work had to be done for the Empirical Experiment. Some of these changes can be incorporated into the development process with minor adjustments.

To save costs and time and at the same time increase user satisfaction, a new bug reporting tool was introduced during the process of the research. In this tool, specifications are defined and test cases are described in detail. There are also 3 different approval steps. On the one hand the developer, on the other hand, the responsible administrator/tester, and finally the customer itself.

In case of incorrect or incomplete specification, it can be adjusted immediately and corresponding new test cases can be created. A new feature can only be deployed if this test run is accepted by the developer, the consultant, and finally the customer. Bugs and also Post-release bugs can be better recorded and subsequently resolved more quickly.

Another application point of this tool is to reduce the Mean Time to Recovery (MTTR). The MTTR is the average time it takes to recover from a product or system failure. This includes the full time of the outage—from the time the system or product fails to the time that it becomes fully operational again.

Since the customer has direct access to this tool, it can be entered immediately if something does not work as it should. In the best case, input parameters or the process that caused the error are also entered. If an error is entered by the customer, the development team is immediately informed and can start with the error correction depending on the urgency.

10.4. Introduction of Deployment Cycles

Moving from irregular unscheduled deployments to regularly scheduled deployment cycles presented several obstacles. Switching from very small deployments almost every day to a large deployment every two weeks could be a real challenge. Almost the same problems were described by Rauf and AlGhafees (2015) in the paper "Gap Analysis between State of Practice & State of Art Practices in Agile Software Development".

Having a fixed time plan for deployments was the goal but at the same time, it was important not to extend the release cadence unnecessarily. A short release cadence is important to get rapid feedback. This first resulted in an overhead of planning and meetings. Since the time was already very tight calculated, it was even more challenging to carry out the planning for the next Sprint and the Product Backlog. In addition, it was difficult to keep the product backlog up to date because of many rapid changes.

By introducing certain deployment windows more pressure was put on the developers which could lead to a reduction in productivity which was also mentioned by Rauf and AlGhafees (2015).

In the initial phase, it often happened that items could not be completed in time and had to be shifted into the next deployment cycle, which in turn led to delays in the schedule.

The next problem which could arise is the customer. It must be communicated that the deployment frequency gets shortened. In the paper, it was mentioned that also customers' unwillingness or their unavailability and the resulting lack of communication could lead to delayed feedback.

The subsequent problem that could arise is the coordination of the developers. In the past, a feature was deployed after its completion, so the feedback cycle was relatively short. However, this was extended by the introduction of sprints. To solve this problem and shorten the feedback cycle again, the plan was to introduce pair programming which in turn resulted in more planning effort and more resource consumption.

Pair programming was not a satisfactory solution for getting immediate feedback, so the company moved to manual code reviews on pull requests combined with continuous integration.

The problems mentioned above must always be considered before switching to time-boxed iterations. However, on the whole, after a few deployment sprints, the switch works very well.

10.5. Further planned improvements

The last section was all about the already modified changes in the development process and the existing features whereas this section is about the future planned improvements concerning the development process.

Some of these changes are already being worked on and preparations are being made to incorporate them into the existing process as quickly and easily as possible. Others need to be considered in more technical detail before they can be implemented.

10.5.1. Creation of a common Error List

Another very easy to implement improvement would be a collected list of common errors which is constantly updated and extended. This allows developers with less experience to use the Error Guessing method and find possible faults. It is also good for experienced ones to see what other developers were testing and with this maybe they find new test cases which they never considered before. This will be a fast and easy way to improve the quality of the Error Guessing test cases and thereby the quality of the code.

10.5.2. Quality Gate on Code Coverage

In the section above it was mentioned that code coverage is considered as the last step before deployment to the production system. To change this, a quality gate can be introduced which only allows newly written or modified code to be deployed to the next higher environment if it has a code coverage of at least 75%.

10.5.3. Integration of Git Commits with Salesforce Cases

In addition to the already mentioned Github actions for Continuous Integration and Continuous Delivery in Section 9.2.3, it could save some effort by introducing automation in connection with commit messages. For example, internal Salesforce cases based on commit messages can be updated. If a commit containing a case number is pushed, this case is automatically set to "In Progress" in Salesforce. If a pull request is created afterward, the status could automatically be changed to "Testing".

To implement this, the commit messages must be used uniformly. The case number must be placed at the beginning of a commit message because messages may be truncated and not displayed in full length by the CI.

10.5.4. Backup with Nightly Synchronization

The development process has already changed very positively regarding the use of version control. With the mentioned script in Section 10.1 it is now possible to synchronize code that has not been changed via Git and to have a change history of it. However, since administrators and other users can also make changes that do not directly affect the code, a different problem arises. This problem could be solved with a nightly synchronization job. The idea is that every night all metadata is retrieved from the live system and pushed to a backup branch. This branch should also be prohibited by direct commits.

10.5.5. Improve Requirements Engineering

During requirements analysis, the needs and desires of stakeholders are identified and developed into an agreed set of detailed requirements that can serve as the basis for all subsequent development activities.³ To avoid delays in the development phase, it must be ensured that the requirements have already been completely formulated and defined before the start of the development phase.

10.5.6. Ensure quality of test cases with Mutation Testing

One of the next steps can be the Introduction of Mutation Testing to ensure the quality of software tests. In mutation testing, some small pieces of code will be changed (mutated). For example, a greater sign gets exchanged with a smaller sign. Another example will be that statements get copied or deleted.

Mutation testing has a high level of error detection but it can be very time-consuming and pricey because of the large number of mutants being tested.

This testing practice should also be automated but keeping in mind that it is very time-consuming it can not be put in the regular Github Actions. It could be possible to implement a new Action that is executed before something can be deployed to the production system on critical tasks.

³Jin, 2018.

11. Conclusion

In the final chapter, the most important results of this master thesis, as well as the chosen approach to achieve them, are summarized once again. In addition, the results are presented with a recommendation for small companies that produce customized Salesforce implementations and want to improve their quality of software and adapt their current software deployment process. In the last section, references are given for further developments on this topic and some recommendations.

The goal of this master thesis was to define a standardized deployment process for small companies which are dealing with Salesforce implementations and to extend this process with suitable test procedures. As expected, the selection of an appropriate testing method has a high impact on the quality of the customized Salesforce implementation as well as a well-defined and appropriate deployment process.

It is becoming increasingly important to focus more and especially earlier in the software lifecycle on the quality of the software. In contrast to former software solutions, it becomes more and more important to bring new versions and updates of the software to the market regularly. Therefore the maintainability and reliability of the software must be always given. Furthermore, errors in the delivered product can lead to fatal consequences, not only in terms of costs and resources but also as a threat to human well-being.

Therefore, this master thesis dealt with the topic of how to increase the quality, especially of custom Salesforce implementations with different testing methods and a suitable deployment process. For this purpose, a gap analysis of the current development process, of a company that focuses on Salesforce implementations, was performed to define the gaps between the current state and the desired process. This analysis showed that there are huge gaps in the standardized use of versioning programs and environment structures. In addition, there was a lack of experience and use of different testing methods and their effective use. Subsequently, solutions were sought to reduce or close these gaps. With an empirical experiment, different black-box test methods were examined and then analyzed for their effectiveness in terms of time, errors found, and applicability.

This led to the result that the use of decision tables can detect errors at a very early stage in the applications lifecycle, for example already in the planning phase. These errors can subsequently be eliminated as quickly as possible with less effort. Boundary Value Analysis can be very useful with just a few test cases. This method can efficiently

find errors. However, BVA used in combination with EP can lead to a reduction in the number of test cases while maintaining the same success rate. The construction of test cases for these two methods can also be started in the planning phase and thus the behavior of possible edge cases can be more closely examined, which can be otherwise easily forgotten.

To exclude conventional errors during the implementation, error guessing can provide a good basis. With this method, many errors are sorted out which usually occur in similar applications. The application of Transition State Diagrams is very similar to the Decision Table method whereas especially during the User Acceptance Test phase attention can be paid to this method. The end-user may have different views and use cases that were not considered during the internal testing phase. Since random testing was the most common method in the company's current testing process, this method was also analyzed and evaluated.

Generating randomized tests is not very cost-effective in terms of time and resources and can therefore be easily integrated into the whole testing process. Random tests can provide a good basis during the development phase. Furthermore, it would make sense, especially concerning Salesforce with their governor limits, to perform stress tests before the deployment phase. The mass of data processed simultaneously is often underestimated and not tested.

By using appropriate test methods in the different phases of the application's lifecycle, quality is emphasized from the very beginning. This automatically leads to a shift from the traditional quality model to the shift-left model, which is the desired state.

To reduce the gap in the standardized use of versioning programs and environment structures, a Git workflow was defined. This workflow is based on the feature branch workflow which was slightly adapted. A standardized environment structure was defined which can be extended as desired. This led to the fact that each branch is assigned to a Salesforce environment. To facilitate synchronization between these, Continuous Integration and Continuous Delivery are used in combination with pull requests. The retrieval of test data and its security was also a point that was considered.

Finally, problems already encountered during the conversion of the deployment process were discussed and possible solutions were proposed. During the changeover, there were also further ideas and suggestions for improvement about quality enhancement, were also considered.

Future Work and Recommendations

Since the tools, frameworks, and techniques for performing software tests are constantly changing and improving due to rapid technical development, a new analysis of test procedures with various tools concerning UI testing and automation of tests could be carried out. Especially tools that enable a custom Salesforce integration could be analyzed in more detail. Furthermore, it would make sense to extend the empirical

experiment to white-box methods to better cover non-functional requirements. Regarding the quality of the generated tests, mutation testing could be a further helpful enhancement.

Quality is a very important issue in software development. Companies that decide to increase it, must be aware that additional resources are needed for these changes. Especially in the initial phase, this can lead to a drastic increase in time resources.

Therefore, each company must decide for itself what should be changed and prioritize these changes. Changes should not be processed in parallel but sequentially. It is important not to change too much at once, otherwise, chaos will result, and possibly everything will remain the same. Changes take time and cannot be implemented from one day to the next. Nevertheless, it is important to set goals and make a plan when which milestones should be reached.

In conclusion, the process of increasing quality is never done; it is a continuous ongoing journey with various goals being achieved along the way. The most important thing is to set goals, focus on them and work continuously to achieve the desired goal in the future.

“Without goals, and plans to reach them, you are like a ship that has set sail with no destination.”

- Fitzhugh Dodson

Appendix

List of Abbreviations

- API** Application Programming Interface. 9
- BVA** Boundary Value Analysis. 42, 44
- CC** Cyclomatic Complexity. 23
- CRM** Customer Relationship Management. 9
- CSV** comma-separated values. 24
- DX** Developer Experience. 13
- EP** Equivalence Class Partitioning. 42
- ERP** Enterprise resource planning. 32
- LoC** Lines of Code. 23
- LWC** Lightning web components. 11
- MTTR** Mean Time to Restore. 21
- Org** Organization. 15
- PaaS** Platform as a Service. 9
- POD** Point of Deployment. 15
- QA** Quality Assurance. 61
- SaaS** Software as a Service. 9
- SDLC** Software development life cycle. 57
- SOQL** Salesforce Object Query Language. 11
- SOSL** Salesforce Object Search Language. 10
- SUT** System under Test. 33
- UAT** User Acceptance Testing. 61
- UI** User Interface. 11

Bibliography

- AlJahdali, Hussain et al. (2014). "Multi-tenancy in Cloud Computing." In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pp. 344–351. DOI: [10.1109/SOSE.2014.50](https://doi.org/10.1109/SOSE.2014.50) (cit. on p. 8).
- Beck, Kent (2000). *Extreme programming explained: embrace change*. addison-wesley professional (cit. on p. 18).
- Beer, Armin and Rudolf Ramler (2008). "The role of experience in software testing practice." In: *2008 34th Euromicro Conference Software Engineering and Advanced Applications*. IEEE, pp. 258–265 (cit. on p. 44).
- Bhat, Asma and S. M. K. Quadri (2015). "Equivalence class partitioning and boundary value analysis - A review." In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 1557–1562 (cit. on pp. 29, 30, 42).
- Cui, Jun (July 2021). "Performance testing for best case." In: *International Journal of Pavement Engineering* (cit. on p. 27).
- Davis, Andrew (2019). *Mastering Salesforce DevOps: A Practical Guide to Building Trust While Delivering Innovation*. Apress (cit. on pp. 14, 18–21, 74, 75).
- Elbaum, Sebastian, Alexey G Malishevsky, and Gregg Rothermel (2000). "Prioritizing test cases for regression testing." In: *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 102–112 (cit. on p. 31).
- Garfinkel, Simson (2005). "History's worst software bugs." In: *Wired News*, Nov (cit. on p. 19).
- Gomes, Ivo et al. (2009). "An overview on the static code analysis approach in software development." In: *Faculdade de Engenharia da Universidade do Porto, Portugal* (cit. on p. 24).
- Graylin, JAY et al. (2009). "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship." In: *Journal of Software Engineering and Applications* 2.03, p. 137 (cit. on p. 24).
- IEEE Standard Glossary of Software Engineering Terminology* (1990). Standard. IEEE (cit. on p. 18).
- Quality management systems — Fundamentals and vocabulary, Requirements* (2015). Standard. International Organization for Standardization (cit. on p. 35).
- ISO 31000:2018(en) Risk management - Guidelines* (2018). Standard. International Organization for Standardization (cit. on p. 18).
- SOFTWARE QUALITY STANDARDS – ISO 5055* (2021). Standard. Consortium for Information & Software Quality (cit. on p. 19).
- ISO/IEC/IEEE 12207:2017 Systems and software engineering — Software life cycle processes* (2017). Standard. International Organization for Standardization (cit. on p. 57).

- Jin, Zhi (2018). "Chapter 2 - Requirements Engineering Methodologies." In: *Environment Modeling-Based Requirements Engineering for Software Intensive Systems*. Ed. by Zhi Jin. Morgan Kaufmann. Chap. 2, pp. 13–27 (cit. on p. 79).
- Jovanović, Irena (2006). "Software testing methods and techniques." In: *The IPSI BgD Transactions on Internet Research* 30 (cit. on p. 28).
- Just, Sascha et al. (2016). "Switching to Git: The Good, the Bad, and the Ugly." In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 400–411. DOI: [10.1109/ISSRE.2016.38](https://doi.org/10.1109/ISSRE.2016.38) (cit. on p. 13).
- Krebs, Rouven, Christof Momm, and Samuel Kounev (2012). "Architectural Concerns in Multi-tenant SaaS Applications." In: *Closer* 12, pp. 426–431 (cit. on p. 8).
- Manchar, Anuradha and Ankit Chouhan (2017). "Salesforce CRM: A new way of managing customer relationship in cloud environment." In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, pp. 1–4 (cit. on pp. 7, 9).
- McCabe, Thomas J (1976). "A complexity measure." In: *IEEE Transactions on software Engineering* 4, pp. 308–320 (cit. on p. 23).
- A Guardian guide to metadata (2013). Accessed: 09.04.2022. URL: <https://web.archive.org/web/20160306145239/http://www.theguardian.com/technology/interactive/2013/jun/12/what-is-metadata-nsa-surveillance#meta=0000000> (visited on 09/30/2013) (cit. on p. 8).
- Nidhra, Srinivas and Jagruthi Dondeti (2012). "Black box and white box testing techniques-a literature review." In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2, pp. 29–50 (cit. on p. 33).
- Patel, Jigar and Ankit Chouhan (2016). "An approach to introduce basics of Salesforce.com: A cloud service provider." In: *2016 International Conference on Communication and Electronics Systems (ICCES)*. IEEE, pp. 1–8 (cit. on p. 10).
- Preedy, Victor R and Ronald R Watson (2010). "5-point Likert scale." In: *Handbook of Disease Burdens and Quality of Life Measures*, pp. 4288–4288 (cit. on p. 47).
- Pullarao, K and K Thirupathirao (2013). "A new way of developing applications in cloud environment using force. com (salesforce. com)." In: *International Journal of Computer Application* 1.3 (cit. on pp. 7, 12).
- Rauf, Abdul and Mohammad AlGhafees (2015). "Gap Analysis between State of Practice and State of Art Practices in Agile Software Development." In: *2015 Agile Conference*, pp. 102–106. DOI: [10.1109/Agile.2015.21](https://doi.org/10.1109/Agile.2015.21) (cit. on pp. 76, 77).
- Reid, Stuart C (1997). "An empirical analysis of equivalence partitioning, boundary value analysis and random testing." In: *Proceedings Fourth International Software Metrics Symposium*. IEEE, pp. 64–73 (cit. on pp. 46–48, 51).
- Rubey, Raymond J, Joseph A Dana, and Peter W Biche (1975). "Quantitative aspects of software validation." In: *IEEE Transactions on Software Engineering* 2, pp. 150–155 (cit. on p. 28).
- Samli, Ruya, Zeynep Behrin Güven Aydın, and Uğur Osman Yücel (2020). "Measurement in Software Engineering: The Importance of Software Metrics." In: *Applications and Approaches to Object-Oriented Software Design: Emerging Research and Opportunities*. IGI Global, pp. 166–182 (cit. on p. 23).

- Sandbox Licenses and Storage Limits by Type* (n.d.). Accessed: 08.02.2022. URL: https://help.salesforce.com/s/articleView?id=sf.data_sandbox_environments.htm&type=5 (cit. on pp. 16, 17).
- Vipindeep, V and Pankaj Jalote (2005). "List of common bugs and programming practices to avoid them." In: *Electronic, March* (cit. on p. 44).
- Westfall, Linda (2016). *The certified software quality engineer handbook*. Quality Press (cit. on p. 23).
- Yin, Junjie (2019). "Salesforce-Usability of Lightning Web Components." In: (cit. on p. 10).
- Yu, Sheng and Shijie Zhou (2010). "A survey on metric of software complexity." In: *2010 2nd IEEE International Conference on Information Management and Engineering*, pp. 352–356. DOI: [10.1109/ICIME.2010.5477581](https://doi.org/10.1109/ICIME.2010.5477581) (cit. on p. 23).

Appendix A.

Questionnaire

General	
Field of Operation	<input type="checkbox"/> Administration <input type="checkbox"/> Development
How many years of experience do you have with Salesforce?	
How many years of experience do you have with programming?	
Error Guessing	
Do you already had experience with this method?	<input type="checkbox"/> I have heard of it but never used it by my own <input type="checkbox"/> yes I used it once <input type="checkbox"/> yes I use it regularly <input type="checkbox"/> never heard of it
On a Scale from 1 to 10 how difficult was i to apply this method?	
How long did it take you to apply this method in total approximately?	
I will use this method again	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
I think with a low number of test cases this method is more effective than random testing	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
Some comments about the method?	

Decision Table	
Do you already had experience with this method?	<input type="checkbox"/> I have heard of it but never used it by my own <input type="checkbox"/> yes I used it once <input type="checkbox"/> yes I use it regularly <input type="checkbox"/> never heard of it
On a Scale from 1 to 10 how difficult was i to apply this method?	
How long did it take you to apply this method in total approximately?	
I will use this method again	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
I think with a low number of test cases this method is more effective than random testing	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
Some comments about the method?	
Transition State Diagram	
Do you already had experience with this method?	<input type="checkbox"/> I have heard of it but never used it by my own <input type="checkbox"/> yes I used it once <input type="checkbox"/> yes I use it regularly <input type="checkbox"/> never heard of it
On a Scale from 1 to 10 how difficult was i to apply this method?	
How long did it take you to apply this method in total approximately?	
I will use this method again	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
I think with a low number of test cases this method is more effective than random testing	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
Some comments about the method?	

Boundary Value Analysis	
Do you already had experience with this method?	<input type="checkbox"/> I have heard of it but never used it by my own <input type="checkbox"/> yes I used it once <input type="checkbox"/> yes I use it regularly <input type="checkbox"/> never heard of it
On a Scale from 1 to 10 how difficult was i to apply this method?	
How long did it take you to apply this method in total approximately?	
I will use this method again	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
I think with a low number of test cases this method is more effective than random testing	<input type="checkbox"/> Strongly Agree <input type="checkbox"/> Agree <input type="checkbox"/> Neutral <input type="checkbox"/> Disagree <input type="checkbox"/> Strongly Disagree
Some comments about the method?	
Task Specifications	
Field of Operation	<input type="checkbox"/> Administration <input type="checkbox"/> Development
How many years of experience do you have with Salesforce?	
How many years of experience do you have with programming?	

Appendix B.

Test Setup

Table B.1 was generated with the Plugin VSCodeCounter for Visual Code.¹ This table shows a statistic analysis of LoC and different programming languages of the code base which was used during the empirical experiment.

Total : 109 files, 4870 codes, 1407 comments, 889 blanks, all 7166 lines

language	files	code	comment	blank	total
Apex	18	1,882	392	207	2,481
JavaScript	34	1,692	952	515	3,159
JSON	10	694	6	6	706
HTML	10	279	37	85	401
XML	28	162	0	28	190
YAML	4	110	12	33	155
CSS	1	40	0	9	49
Shell Script	1	5	0	1	6

Table B.1.: Code Base Analysis

Source: Generated with VSCodeCounter from own Code Base Project

B.0.1. Task Descriptions

Task 1

To reserve a car, it should be possible to select one from a list of cars.

Depending on the selected customer (account) who wants to rent the car, only certain cars should be displayed. If the customer has children, only cars where a child seat is available and which are approved for child seats (see Task 2 ≤ 110 PS) may be displayed. Furthermore, customers under 30 may only rent cars with less than 111 PS and only rent them without a child seat, otherwise an error message should be

¹<https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter>

displayed. If the rental date is between 1.11 and 1.3 (winter) the car must have all-weather tires. Cars with all-weather tires may only be driven in winter. Furthermore, only cars that have not yet been reserved or rented should be listed.

Pick up and drop off stations must be selected. If the car is picked up and returned at the same station you will get a discount of 10The booking can only be completed if the customer has a subscription. Otherwise, the customer should be informed that the subscription fee will be automatically added to the invoice. With a checkbox it should be possible for the customer to reject this and cancel the booking or to accept and thus make the booking successfully. If the customer accepts, the field Subscription existing should also be adjusted on the customer record.

If a booking is completed, a rental for this account should be created automatically. The rental should contain the data of the car, start and end date and the calculated price.

Task 2

It should be possible to create a car (Vehicles). Depending on the horsepower, the subscription class needed for billing should be set automatically, taking into account that cars with more than 240 PS cannot be created. Cars with child seats must not have more than 110PS.

- 50 - 86 PS small
- 87 - 110 PS medium
- > 111 large

[B.1](#) shows the implemented Screenflow which was the basis for Task 1 of the experiment.

Figures [B.2](#), [B.3](#) and [B.4](#) are Screenshots of the applications.

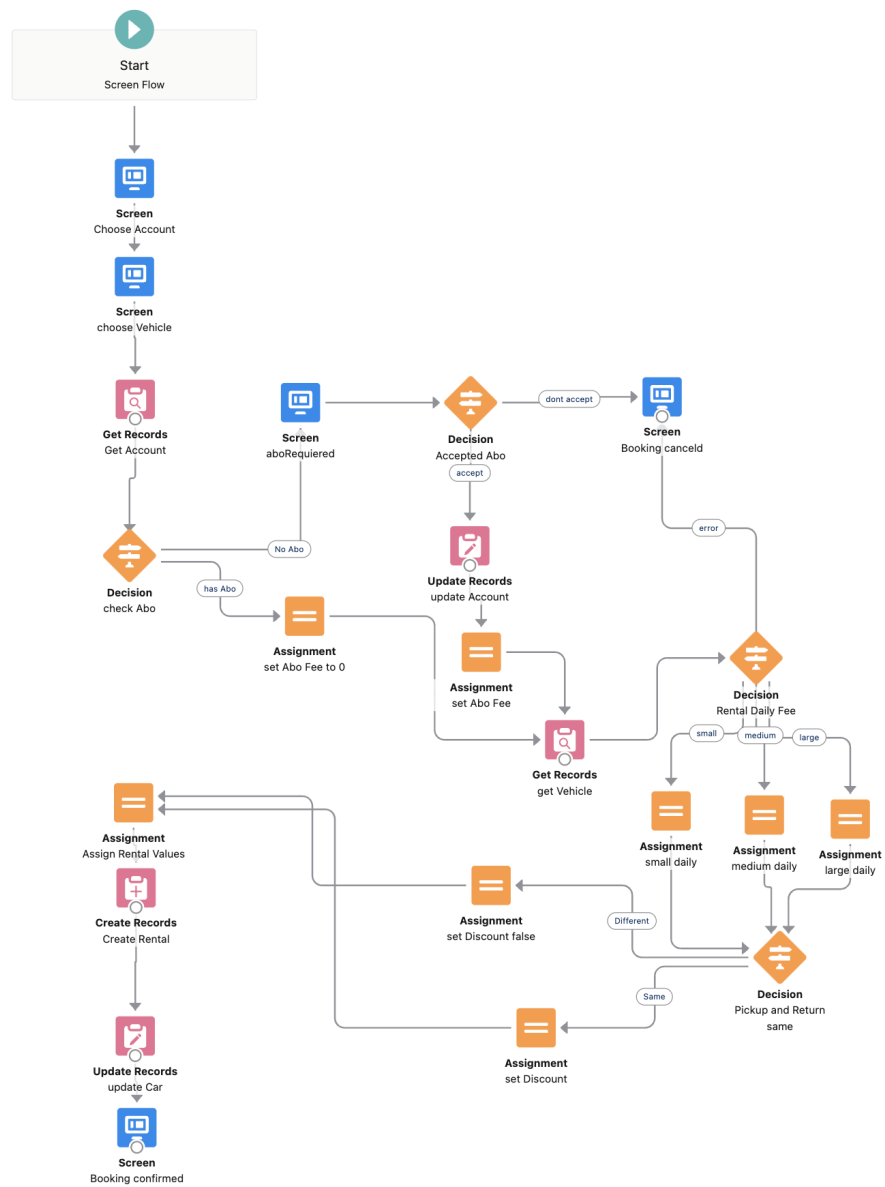


Figure B.1.: Screenflow for booking a vehicle
Source: own representation

Book Rental

* Account

* Start Date

End Date

Pickup Station

Return Station

Next

Figure B.2.: Screenshot Task 1: Enter Booking Details
Source: own representation

Book Rental		
VEHICLE NAME	ALL-SEASON TIRES	CHILD SEAT AVAILA
3er GOLF Rabbit	false	true
MAZDA 5	false	true
Car CM 2 (C, No AST, 50)	false	true
AT Vehicle Small Summer only Child	false	true
AT Vehicle Child 100	false	true
PEUGEOT 206	false	true

Previous Next

Figure B.3.: Screenshot Task 1: Choose available Vehicle
Source: own representation

New Vehicle

Information

* Vehicle Name

Seats

All-season tires

☐

* PS

Child Seat Available

☐

Owner

Anna Haupt

Color

--None--

Rental Status

free

Abo Class

--None--

Cancel

Save & New

Save

Figure B.4.: Screenshot Task 2: New Vehicle Creation
Source: own representation