Thomas Kohl, BSc

Developing a Concept for the Introduction of Behaviour Driven Development in Agile Development Teams for E-Commerce-Platforms

Master's Thesis

Graz University of Technology

Supervisor: Slany, Wolfgang, Univ.-Prof. Dipl.-Ing. Dr.techn.

Institute of Software Technology

Graz, July 2019

Statutory Declaration

Eidesstattliche Erklärung

Abstract

Continuously changing requirements are one of the biggest challenges software development teams face in their daily business. This applies especially for teams focussing on the development of web applications, such as E-Commerce-Platforms. Research has shown that the introduction of Behaviour Driven Development in the development process is a possible solution to prepare a team for any upcoming changes. In this thesis a concept is created to integrate Behaviour Driven Development in teams of an organisation focussing on such E-Commerce-Platforms. The proposed concept puts emphasis on the improvement of the requirements engineering process as well as on the verification of requirements over the whole life cycle of a software project. Based on a literature review on these two focal points a selection has been made tailored to the needs of the organisation. This led to the introduction of a workshop technique as well as a test framework as a basis of the concept. The proposed concept serves as a foundation for a bigger process change in the selected environment. First efforts to introduce this concept show that it can improve the development process in software development teams. In particular, the analysis of complex requirements is enhanced. In addition, further possible effects of the concept's introduction are addressed.

Kurzfassung

Ständig ändernde Anforderungen sind eine der größten Herausforderungen für Softwareentwicklungsteams in ihrem Alltag. Dies gilt insbesondere für Teams, die sich auf die Entwicklung von Webanwendungen wie E-Commerce-Plattformen spezialisieren. Untersuchungen haben gezeigt, dass die Einführung von Behaviour Driven Development eine mögliche Lösung ist, um ein Entwicklungsteam auf jegliche bevorstehende Änderung vorzubereiten. In dieser Arbeit wird ein Konzept entwickelt, um Behaviour Driven Development in die Softwareentwicklungsteams einer Organisation zu integrieren, die sich auf solche E-Commerce-Plattformen spezialisiert hat. Das vorgeschlagene Konzept legt den Schwerpunkt auf die Verbesserung des Prozesses zur Aufnahme von Anforderungen sowie auf Möglichkeiten, wie Anforderungen über den gesamten Lebenszyklus eines Softwareprojekts verifiziert werden können. Basierend auf einer Literaturrecherche zu diesen beiden Schwerpunkten wurde eine Auswahl getroffen, die auf die Bedürfnisse der Organisation zugeschnitten ist. So bilden eine Workshop Technik sowie ein Testframework die Basis für das Konzept. Dieses Konzept bildet die Grundlage für eine größere Prozessänderung in der ausgewählten Umgebung. Erste Versuche, das Konzept einzuführen, zeigen, dass es Teile des gelebten Prozesses in den Softwareentwicklungsteams verbessern kann. Insbesondere die Analyse komplexer Anforderungen wird dadurch verbessert. Zusätzlich dazu werden weitere Auswirkungen, die die Einführung des Konzepts auslösen kann, behandelt.

Contents

List of Figures

List of Listings

List of Tables

1. Introduction

In the rapidly changing environment of software projects, there is only one constant. This applies especially for software projects that are focussed on web development, like E-Commerce-Platforms. This constant is continuous change. Whether it comes to customer's requirements towards the software itself, staff changes to the development team or all kind of influences from the outer world, such as changing laws or regulations. A software development team must be able to cope with all of them and must be able to address them in a timely and decent manner.

Changes to the team set up and constantly changing requirements pose enormous challenges. Therefore, it is immensely important to have a clear overview of the current status of the software and, of course, if its requirements are still met. Not only existing requirements represent obstacles for a software development team, but also new requirements need to be addressed adequately. Thus, a team will always have to cope with one of the hardest challenges: people understanding the same problem differently. To summarise, the following main hurdles need to be addressed:

- How can a development team effectively handle and communicate new requirements, in a way such that, all different stakeholders have the same understanding?
- How does a development team manage existing requirements effectively to be able to respond if the team or the requirements change?
- How can a development team be sure that the existing requirements are still met after changes have been made?

Thus, for a software development team, it is essential to have a common understanding on all levels, to be able to succeed. Facing such a typical problem during development, Dan North, with the help of others, realised, that after he started seeing the requirements as a behaviour the application must fulfil, many of his problems vanished. Furthermore, he concluded that he was trying to define a ubiquitous language to analyse such problems without even thinking in that direction. By defining a template to formulate examples of those behaviours, he was able to communicate the requirements effectively. This technique provides an efficient way of verifying that the behaviour – the requirements – are met. He named this technique *"Behaviour Driven Development"* [1].

1. Introduction

Gojko Adzic focussed in one of his works *„Specification by Example"* on various case studies and experiences from multiple teams, that tried to implement such a technique. Almost all investigated case studies showed that applying these practices to their project had improved the process, the documentation of the product as well as the quality of the delivered product itself in comparison to the previous approaches. Gojko Adzic also showed that there are already various techniques sharing the same key principles. According to his understanding, it can be argued that the following terms can be used interchangeably [2]:

- Behaviour Driven Development
- Specification by Example
- Acceptance Test Driven Development
- Story Test Driven Development

The techniques named above do have numerous similarities. Therefore, it does not make sense to distinguish between all of them but rather focus on the underlying base concepts [2]. With that in mind, the term *"Behaviour Driven Development"* will be used for any following reference to those core principles.

With this thesis, a concept is established on how a software development team focussing on E-Commerce-Platforms might be able to deploy Behaviour Driven Development to address all the stated well-known problems. The focus is set on two parts of Behaviour Driven Development. Firstly, providing a proposal on how software development teams could extract the needed requirements. As a second part, the concept describes how these requirements can be used to drive the development process and ensure a continuous verification of the extracted requirements.

By applying this concept an agile software development team should be able to implement the desired functionalities more efficiently. One of the reasons for this is that all requirements are supported by examples. Hence, the chance of introducing ambiguous requirements is decreased. Additionally, using examples as executable tests will give the software development team the needed security to respond to any upcoming change with the required self-confidence. The executable requirements would indicate immediately, if any existing functionality is broken. Moreover, this concept should help a development team to further improve its development process and thereby making it more efficient. Additionally, the increased self-confidence should enable the development team to reduce the needed time to deploy new functionalities into production.

In the following section, different well-known software development process models and some of their practices are reflected to point out common weaknesses and problems. Furthermore, the disciplines of requirements engineering, testing and the characteristics of Behaviour Driven Development are outlined. After the basics, the environment, in which the concept should be introduced, is described. The following two chapters concentrate on different techniques for the defined focal points: requirements engineering and tools and frameworks that could be used to support the key elements of Behaviour Driven Development. At next, the concept itself is introduced. This approach is divided into two parts, corresponding to the points of interest as stated earlier. The first part of it focuses on improving the way requirements are discovered and managed. Subsequently, the second part is centred on how the requirements can be automated sustainably. In a further section, possible ways of integrating the concept in the desired environment are described. After that, the findings are discussed. Finally, the thesis is concluded and an outlook for future research work is given.

2. Basics

In the following section, some commonly used processes and methodologies are introduced. The focus is set on different software development process models, requirements engineering, a short overview of software testing, how requirements could be used as test cases and the core principles of Behaviour Driven Development. These topics are among the most important disciplines of software development, building the foundation that needs to be understood if Behaviour Driven Development should be introduced in a software development team.

2.1. Software Development Process Models

There are numerous different software development process models in usage. According to Ruparelia, such a model describes the organisation and structure used for developing software. He stated that classifications for software process models could be done based on its nature to repeat or not repeat any of its stages. Such a model can either be linear, iterative or a combination of both. Linear models are structured sequentially, meaning that finishing one phase of the model leads to the next one. Iterative models repeat certain or all steps of the process model. They aim to use the knowledge and results of the last iteration as a basis for further improvements [3].

2.1.1. Linear Software Development Process Models

Based on that classification, one of the oldest and most famous, in its pure form a completely linear, software development process model is the *"Waterfall Model"*. Alongside that, the *"V-Model"* is described as the two linear software development process models.

2.1.1.1. Waterfall Model

Although it wasn't called *"Waterfall Model"* initially, its basic idea was already described by Winston W. Royce in 1970. He mainly described his experience and the problems he encountered in various big software projects for spacecraft mission planning and commanding as well as for analysing flights. In Figure 1, the typical development life cycle of the Waterfall Model is shown. At first, requirements are determined. Based on these requirements, the analysis phase is conducted. After the design phase, the actual implementation is done with a subsequent testing phase. After the testing phase, it is possible that feedback arises, that leads to changes of program design or on the requirement level. Finally, after testing, the product is delivered [4].

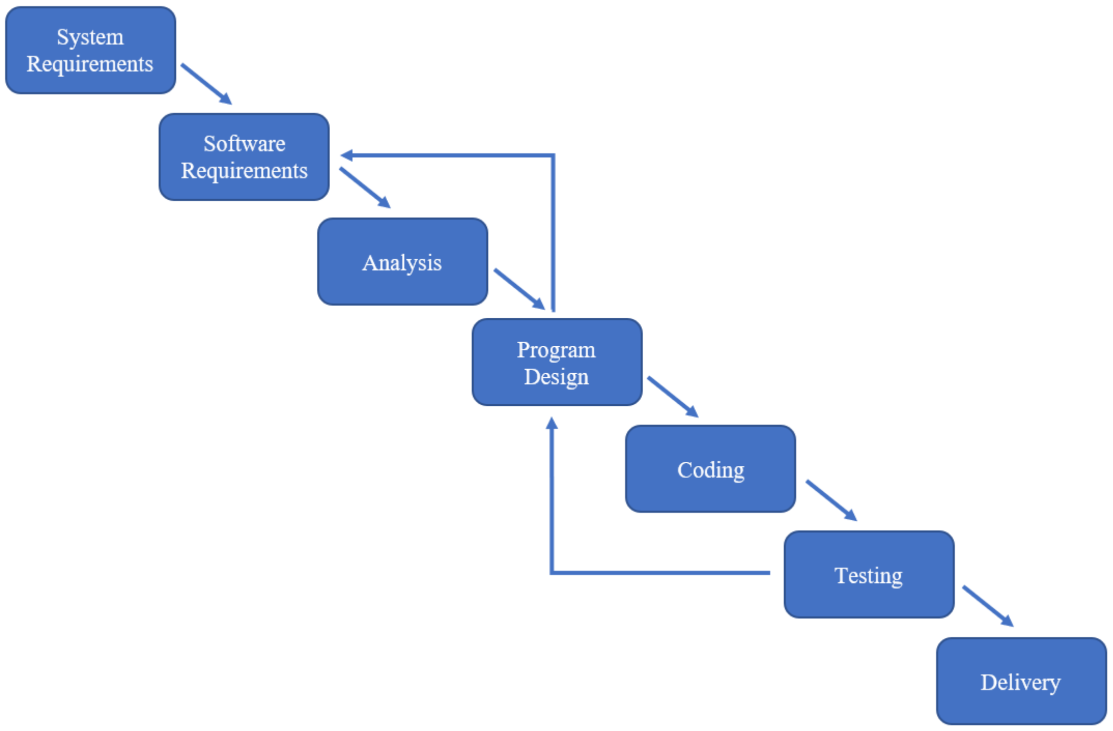


FIGURE 1: WATERFALL MODELL [4]

Royce also described possible adaptions to that basic approach. These adaptions, as listed below, should mitigate some limitations of the pure sequential approach [4]:

1. Program Design comes first
2. Document the Design
3. Do it twice
4. Plan, Control and Monitor Testing
5. Involve the customer

These five adaptions to the Waterfall Model show that already from an early stage onwards, it was known that late feedback could be a driver for costs in software projects. Involving the customer throughout all phases of the project might limit the risk of delivering unusable or unwanted software. Especially, developing a first pilot version of the project can help to reduce the risk even further [4].

The Waterfall Model was also subject of an investigation of Laplante and Neill in 2004. They investigated best practices and myths concerning software development. According to them, the Waterfall Model comes from a time in which requirements rarely came from different stakeholders and changed very infrequently. Therefore, its application had been widely successful. However, in most of today's software development projects, changing requirements are omnipresent, and the Waterfall Model is not suited as it is inflexible towards change [5].

Similarly, Ruparelia analysed various software development process models, including the Waterfall Model, and tried to categorise different process models based on the type of application that should be developed. He chose following categories:

1. Applications providing only back-end services to other applications, such as a database application
2. Applications wrapping around business logic to provide a service to an end-user application, such as an Application Programming Interface
3. Applications focusing on direct end-user interaction by providing a Graphical User Interface, such as a web application

Most of the linear process models can only be classified as suitable for the first two categories of this list. He concluded that breaking a project into smaller parts and deliver multiple releases can be a solution to an omnipresent change of scope. The best way to do this is by applying agile software development process models as the chosen way of working [3].

2.1.1.2. V-Model

The before mentioned inflexibility of the Waterfall Model is tried to be mitigated in several adaptions of it. One prominent representative of that group is the V-Model. The different phases of this linear software development process model are arranged in a V-shape. These phases are read from the top left to the bottom and back to the top right, resulting in a folded version of the Waterfall Model. On the left side, the steps of requirements engineering and design are pictured. At the bottom, the actual implementation of the application is described. The following upward stream includes a phase for each one on the left as verification. In Figure 2, it is illustrated that the Decomposition and Definition stream oppose the Verification and Integration Stream [6,7].

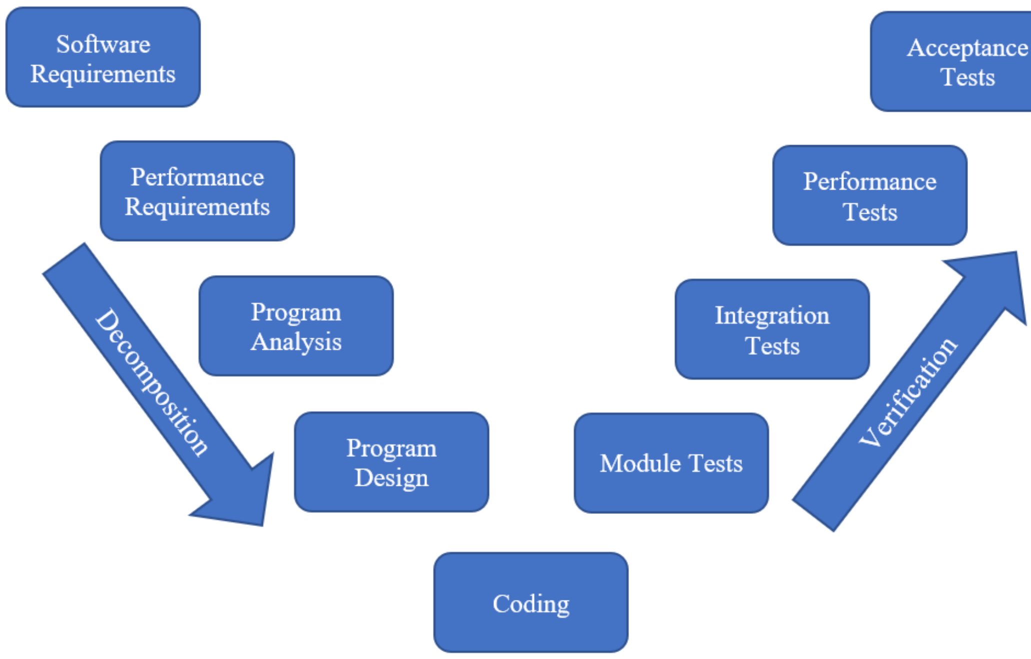


FIGURE 2: V-MODEL [6]

2.1.2. Agile Software Development Process Models

In the world of agile software development, there is no way around its most basic principles: The Manifesto for Agile Software Development by Kent Beck et al. They agreed on 12 principles to formulate a definition of what agile development is about, from which the quintessence is even shorter [8]:

“Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan.” [8]

Based on these principles, several agile software development process models have been implemented and reviewed in different fields of software development. In a survey by Barabino et al. several commonly used practices were investigated. According to them, an agile software development process model is highly suited for most web applications due to vague or changing requirements. One of the survey's findings was that the most commonly used agile approach was Scrum, followed by Extreme Programming. Another interesting observation was that most of the respondents used User Stories to organise and communicate requirements. Whereas the use of automated acceptance tests was low. Only about 21% of the respondents confirmed that this practice was performed regularly [9].

Additionally, the approach of agile methodologies is no longer limited to software development alone. Moyo et al. showed that an agile method, in their case Scrum, may as well be used in numerous other fields or research problems. They applied this idea to develop a simulation of Alcohol Consumption Dynamics. According to them, the decision to use an agile methodology was made because of two main reasons. Firstly, the used research data was not gathered for their use case. Thus, they not only had to deal with the parameterisation of their research data but also changing and balancing the used modelling requirements, meaning that they had to deal with changing requirements. Secondly, their research project had a fixed time-constraint of twelve months and a small development team. Therefore, they wanted to be able to deliver a working prototype at an early stage which could evolve incrementally. Moyo et al. added that because of choosing an agile way of working, they have been able to prioritise their work properly and hence deliver first results at the beginning of the project. They concluded that more researcher should think of adopting such methodologies [10].

In the next sections, some of the most commonly used agile software development models are introduced to provide overview of these practices in detail.

2.1.2.1. Extreme Programming

Extreme Programming was introduced by Kent Beck in his work *"Extreme Programming Explained"* [11]. He recognised the basic problem that all software development projects have in common: risk, such as:

- Not being able to deliver the software in time
- To deliver the software in time, but with numerous issues
- To deliver software not solving the intended business problem

He pointed out, that for software projects, no matter which domain, there are four control variables, which are all interrelated:

- Cost
- Time
- Quality
- Scope

The most noteworthy thing about the mentioned variables is, according to him, that all involved stakeholders, such as developers, customers and managers, are aware of their presence. If these variables were changed deliberately, it would have a direct effect on all the others. Hence, he added that there is no simple relationship between them. However, Kent Beck argued that there is one variable that, if managed properly, would leave the management and the customer with control over cost, time and quality: scope. As scope usually varies a lot over the lifetime of a software project, the risk of a project failure could be mitigated by getting early feedback and implementing the most important features first. Regarding changing requirements, Beck stated, that the cost of a change in a program will be exponentially higher the later it will be done, as seen in Figure 3 [11].

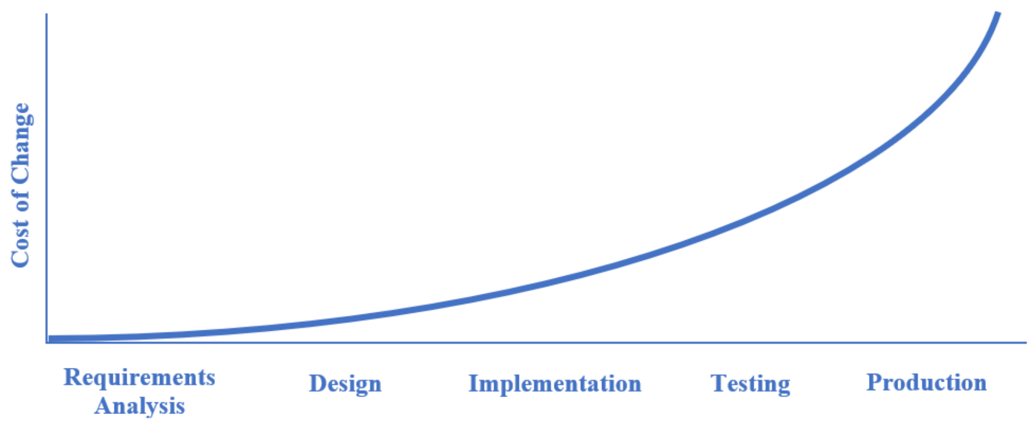


FIGURE 3: COST OF CHANGE [11]

Beck added an additional assumption, that could influence the cost of change. He stated that the cost of changing parts of the application would not behave in that way if a few simple rules were followed. Simplifying the design, writing automated tests to have confidence in the code and constant refinement of that design and tests could lead the team to not be afraid of making changes when they must be done. With that assumption in mind, Beck stated, that managing a software project is like learning how to drive a car. The trick is not only to head in the right direction but paying attention and make small or big corrections to stay on track. Based on that he stated some core values for Extreme Programming [11]:

- Communication
- Simplicity
- Feedback
- Courage

Based on these values, Kent Beck introduced a system consisting out of twelve practices, that, if used correctly, is a powerful tool to limit the risk of project failure:

1. **The Planning Game**: plan the next release and respond to change by updating the plan
2. **Small Releases**: release a simple system as soon as possible
3. **Metaphor**: make a story to guide the whole team
4. **Simple Design**: the system contains only the parts it needs; no unused functionalities
5. **Testing**: developers and customer write automated tests to ensure quality
6. **Refactoring**: if there is a way to restructure code to make it easier to understand, do it
7. **Pair Programming**: two developers code together - always
8. **Collective Ownership**: nobody should hesitate to change any part of the code

9. **Continuous Integration**: after each change, the system is built and tested
10. **40-hour week**: avoid overtime at all cost – stressed developers are inefficient
11. **On-Site-Customer**: the customer should always be available to answer questions
12. **Coding Standards**: communication through the code should be evident

He added that all the mentioned practices support each other. If any of these were not performed by a team, it would bring the whole process out of balance. As another key aspect of Extreme Programming, Kent Beck added that decisions should be taken by those who are suited to do so, that means business stakeholders should take responsibility for business decisions, and developers should take responsibility for technical ones. Thus, business decisions should drive the project and developers should inform their business colleagues about costs and risks of technical decisions. In the end, Beck concluded that all software development projects are based on fear, but Extreme Programming prepares the development team with the ability to respond to any change making it a powerful approach [11].

2.1.2.2. Scrum

According to Ken Schwaber and Jeff Sutherland, Scrum is not a specific process, but a framework that allows the application of different processes and techniques. The core of Scrum is a small independent team that could collaborate with other teams. The idea of this agile methodology is, that the process itself evolves out of the experience of the software development team. Therefore, three pillars have been chosen to describe how a process should evolve inside the Scrum framework [12]:

- **Transparency**: the process itself has to be communicated and visible to all members of the team
- **Inspection**: Scrum artefacts must be inspected regularly to indicate progress or stagnancy
- **Adaption**: if an inspector finds an undesired development the process has to be adapted

For the Scrum Team itself, only the following roles were prescribed by its originators Schwaber and Sutherland [12]:

1. **Product Owner**: manages the product backlog, a continuously changing list of all work packages, by ordering and explaining the items in it

2. **Scrum Team**: a self-organising development team with the required skills to complete all upcoming tasks
3. **Scrum Master**: plays the role of a moderator and coach that supports the Product Owner and the Scrum Team to live and evolve the process

The momentarily most important product backlog items are chosen and planned inside a Sprint. A Sprint is a fixed time frame of a month or less with a specified goal in which the development team tries to complete all previously planned items, the Sprint backlog. The scope of this Sprint backlog can be defined based on estimations of the product backlog items. These estimations usually indicate if the development team can finish the product backlog item in the upcoming Sprint. If this is the case, it can be planned. During the Sprint, Scrum prescribes a set of meetings that should help the team to stay on track [12]:

- **Sprint Planning**: The Sprint backlog is filled with the most important items of the product backlog that are ready to be implemented
- **Daily Scrum**: A short daily meeting to ensure proper communication and identifying risks and problems during the Sprint
- **Sprint Review**: A meeting to present the outcome of the last Sprint to get feedback from different stakeholders
- **Sprint Retrospective**: The Scrum Team sits together and tries to find ways to improve the process and collaboration inside the team

Finally, Ken Schwaber and Jeff Sutherland concluded that it is possible to use only parts of Scrum as well, although it should then not be called Scrum anymore as it works best if all practices are performed as recommended in *“The Scrum Guide”* [12].

2.1.2.3. Kanban

In his work *“Getting started with Kanban”*, Paul Klipp summarised the essence of Kanban as a tool used to guarantee an optimal flow of work in a software development team [13].

The idea behind this can be traced back to the terms *“Lean”* and *“Just in Time (JIT)”*. According to Ebert et al., these techniques were used as a production management process in the beginning. Toyota used such an approach to be able to deliver small batches just as they were needed, which resulted in drastically reduced working hours and improved quality. All of this was achieved by changing the production flow from a push to a pull mentality [14].

Klipp added that there are only three rules to be considered when working with Kanban [13]:

- **Visualise Workflow**: Visualise how the work packages flow through the process until they are done, no matter how complex the workflow might be – usually a Kanban Board is used for that purpose. An example is pictured in Figure 4.
- **Limit Work in Progress (WIP)**: Increase efficiency by concentrating on one task at a time, because working on different things simultaneously has proven to be slow, like Klipp said: *"Get more done by doing less"* [13].
- **Measure and Improve Flow**: Make small adjustments to the process if needed, so that identified bottlenecks can be dissolved.

Two of the most important indicators to identify bottlenecks in a process are lead time and cycle time. The lead time gives an indication on how long a work package needed from the time of request until it was done. Whereas the cycle time gives information on how much time is needed for a work package to be finished from the moment on somebody started working on it [13].

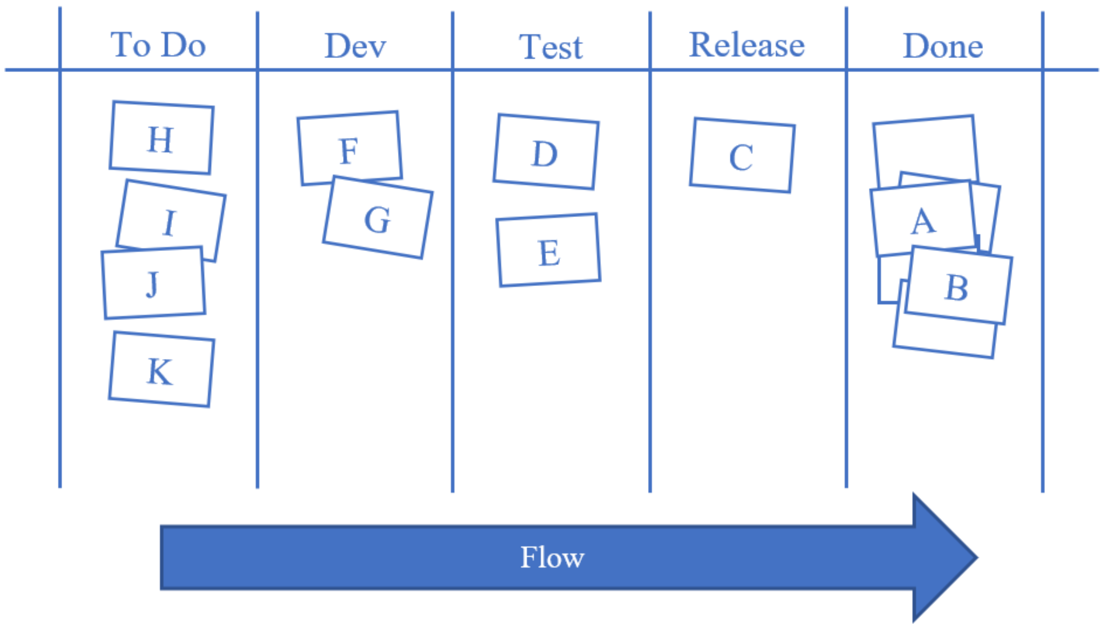


FIGURE 4: EXAMPLE OF A KANBAN BOARD [15]

2.1.2.4. Agile Software Development: Summary

To the question of which of these frameworks or guidelines should be used for a software project, no definite answer can be given. According to Kniberg and Skarin, none of the named methodologies are the answer to all problems, but they provide interesting guidelines and constraints. Moreover, they stated that all those techniques can be mixed. Kniberg and Skarin cannot imagine a Scrum Team not using most of the techniques Extreme Programming prescribes or a Kanban Team that does not use Dailies, which is a Scrum practice. Thus, it is possible to use all techniques of either methodology suitable for the specific team and use case. However, it should be considered that in case something does not work as intended, it should be possible to be changed. As already mentioned above, none of the named methodologies provides a definite solution for any case. The goal should be continuous learning from experience. Kniberg and Skarin summarised this by saying [15]:

“The only real failure is the failure to learn from failure.” [15]

According to Olsson et al., all different software development process models can be classified towards a step on their *“Stairway to Heaven”*. They stated that an organisation must split up large development groups into smaller teams, which focus on smaller junks in short development cycles instead of big components, to get from the first step *“Traditional Development”* to an *“Agile Organisation”*. Such a transition could be from the Waterfall Model to Scrum or any other agile software development process model. *“Continuous Integration”* as the next step implies that automated tests are developed and executed against the code base. *“Continuous Deployment”* is the next step on this stairway. It requires short feedback cycles that involve product management as well as customers. This step implies not only that the code is checked automatically but also is continuously delivered. In the last step a switch of thinking can be found. The deployment to the customer's system is seen as the starting point, and the requirements of the product are extracted directly out of user feedback of the current system. This means that the requirements evolve in real-time instead of being declared before the start of the development. The name of this last step is *“Experiment Systems”* [16,17].

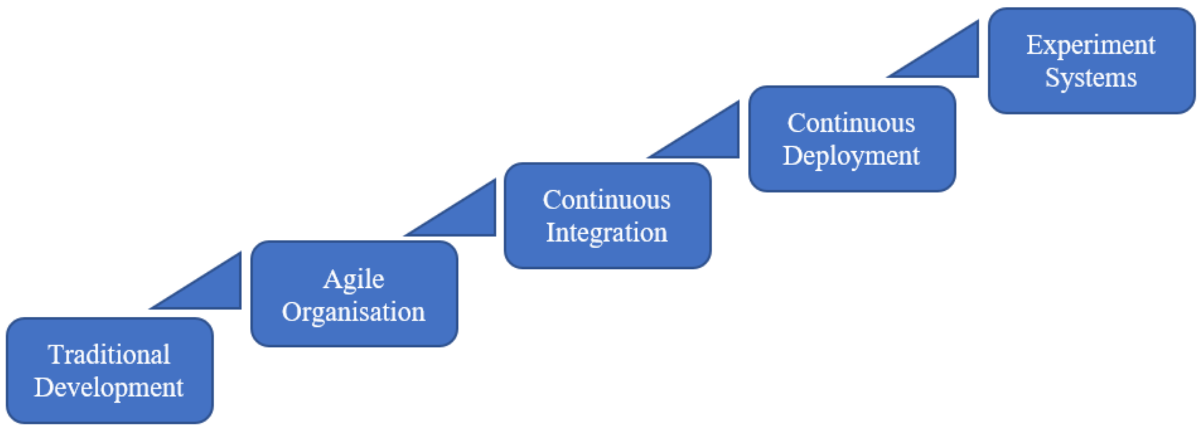


FIGURE 5: STAIRWAY TO HEAVEN [17]

Thus, according to Olsson et al. adopting to an agile methodology is only one step on their *"Stairway to Heaven"*, as illustrated in Figure 5. This is a crucial way of thinking about agile software development process models, as none of them provide a definite solution. However, agile methodologies provide guidance to a possible solution that must be discovered for each team individually, no matter which of them is applied. Moreover, even after a team has found a working process, there still is a long way for improvement in order to climb all steps on the shown stairway [17].

2.2. Requirements Engineering

Gojko Adzic replicated an example by Weinberg and Gause and focused on different interpretations people can have about the same topic [18]. He handed out cards showing the star pictured in Figure 6 and asked the attendees how many points the star had [19].

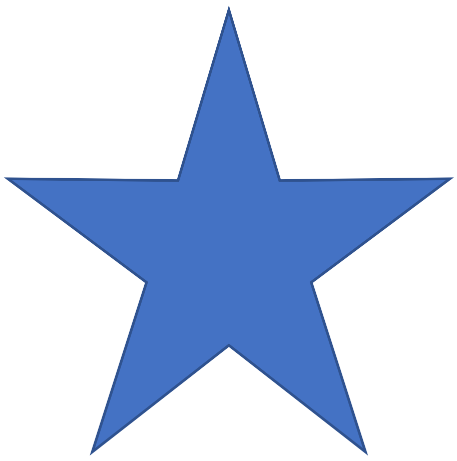


FIGURE 6: COUNTING THE POINTS OF A STAR [19]

Adzic pointed out that most of the people were certain to have chosen the correct amount of points for the shown card. The majority of them said that the star does have ten points. However, some of the participants counted a different number of points, e.g. five points. Adzic showed with this short example that even if the people were certain to have chosen the correct answer, not all of them came to the same conclusion. This reflects the situation in most software projects – different stakeholders talking about the same requirements despite having a completely divergent understanding of them [19].

In general, according to Nuseibeh and Easterbrook, requirements engineering is the process of how to formulate the needs of different stakeholders in a way, that it can be used for the subsequent development steps. In addition to that, this process is an essential part during the software development process itself. It is a multi-disciplinary and human-centred process, which is usually underestimated when it comes to its effects on the overall success of a software project. Moreover, requirements engineering is not only a process to initially capture functional and non-functional requirements, but it should also cover the communication of these requirements to all involved stakeholders to find consensus between all of them. This means they came to a similar conclusion as Gojko Adzic, as shown above [19,20].

There are numerous approaches that try to make the process of requirements engineering easier and more understandable. Reichart et al. proposed an approach that should support the process of capturing requirements by applying a task-model-based approach to describe features. They concluded, that the analysis of user-centred requirements is supported best by visualising functional requirements with their proposed solution [21].

According to Eric S. K. Yu, using tools to aid the requirements engineering process is inevitable, but using a fitting tool at the right time is challenging. He pointed out that modelling techniques or different visualisations are often used to aid engineers with their tasks. However, he also stated, that most of these techniques are used in *"late phase"* requirements engineering tasks with a focus on completeness, precision and consistency. While those *"late phase"* tasks get much attention, he noted that the *"early-phase"* requirements engineering tasks are being neglected too often. Nevertheless, these *"early-phase"* tasks or activities represent some of the most important questions and therefore should be asked before starting with software development. They describe, how the planned system should behave to meet initial business goals from a high-level perspective. Additionally, the *"early-phase"* tasks could help the team to understand why a software or feature should be built in the first place. Yu concluded, that understanding the initial reason and using the right techniques and tools during requirements engineering activities might bring more structure to the process of discovering the requirements itself [22].

In contrast to such an approach, there is a standard documenting best practices or guidelines how a requirement specification document should be structured. Additionally, this standard prescribes how the requirements should be written. This standard is named IEEE-830-Standard [23]. It recommends writing functional requirements in a specific form: *"The system shall ... "*. Moreover, this standard also recommends to be as concise as possible with the requirements before the start of the implementation and introduces guidelines to end up with requirements in a decent quality [24].

There are various other ways of how requirements could be defined and processed, but as already mentioned previously in chapter 2.1.2, the most commonly used approach to specify requirements for agile software development process models is User Stories [9].

2.2.1. User Stories

Mike Cohn published an article examining the advantages of using User Stories in comparison to well-established requirement specification practices, like the IEEE-830-Standard mentioned before. At first, he stated, that the concept of User Stories was introduced by Extreme Programming and has ever since been adopted widely in all agile methodologies. A User Story is a short description of a goal about a certain value the software should fulfil from the view of the user or the end user of the product. Usually, these stories are written on so-called story cards, which are used for further discussions [24]. According to Ron Jeffries, a User Story should always consist of three components [25]:

1. Card
2. Conversation
3. Confirmation

Cohn built his understanding of this alliteration and added that the card is not the most important thing of a User Story. It should be used as a reminder for subsequent planning and as a central point for conversations around the intended goal of the story. As a result of these conversations a documentation should be created holding the confirmations of the story. These documented confirmations should afterwards be used to verify whether a story is completed or not. He concluded that a User Story should encourage verbal communication, to ensure a common understanding of the upcoming task between all involved stakeholders [25].

Antony Marcano pointed out that User Stories are often misinterpreted in a way that they describe what the user will have and not what the user will be able to do as soon as the User Stroy is implemented. In other words, a specific solution or thing is described instead of telling what the user wants to do with it. Antony Marcano pointed out that there is a difference between a feature and a User Story [26]:

“When what we want is “a thing” then what we are describing is not a user story – it is a feature.” [26]

By using examples to highlight the difference between a User Story and a feature, Antony Marcano made it clear that when using a User Story, there is no limitation to a specific solution. In contrast, features clearly describe one possible way how the goal should be achieved. This difference makes it obvious that the flexibility of achieving a certain goal is decreased if the description depicts a feature instead of a User Story. In Table 1 some examples are pictured to highlight this difference [26].

User Story	**Feature**
As a cook, I want to prepare hot food for the family, so that we can safely enjoy various fresh foods.	As a cook, I want a kitchen, so that I can prepare food.
As a parent, I want to sleep comfortably with my partner, so that we can keep each other warm.	As a parent, I want a master bedroom, so that I can sleep comfortably with my partner.
As a car-owner, I want to park somewhere safe, so that I can keep my insurance premiums low.	As a car-owner, I want a garage, so that I can park my car off the street.
As a microblogger, I want to find out as soon as I am mentioned, so that I can choose to respond while topics are current.	As a microblogger, I want a notifications screen showing my last notification, so that I can find out as soon as others interact with me.

TABLE 1: COMPARISON OF USER STORIES AND FEATURES [26]

Gojko Adzic described the same problem in his work *“Bridging the Communication Gap”*. According to him, requirements already form a proposed solution instead of underlying *“why”* the solution is needed or *“what”* goal should be fulfilled with it. Thus, this is not only a problem inherent to User Stories but seems to be a more general problem. Therefore, he added that it would be a good practice to share as much detail about the *“why”* with all involved people, since anybody could suggest a simple solution to the given problem [19].

The so-called Connextra Template describes a specific way in which User Stories should be written. This template is structured in the following way [27]:

As a [someone]
I want to [do something]
So that [some result or benefit]

Antony Marcano stated, the motivation behind this template is to support all involved stakeholders to start a conversation about the goal, which should be achieved with a User Story. This is accomplished by giving a starting point for a short story to explore what the user would want to do. Several variations of this template have been created over time, such as the following one [27]:

As a [role]
I want [feature]
So that [benefit]

Marcano argued that these variations were introduced by teams because it was easy for them to adapt to their User Story when they were still working with features, which described requirements. As it can be seen in Table 1, a description of a feature is also possible with the same template. This makes it even harder to use the Connextra Template properly for a User Story only. Thus, according to Antony Marcano, teams working with *"Features dressed as User Stories"* do have a fundamental misconception of using User Stories in the right way. Additionally, he also agreed with Jeff Patton, who said, User Stories are not only a different way to write requirements but are a different way to work [27,28].

The Connextra Template is not the only pattern that tries to define how a User Story should be built and what it should contain. Another concept describing this is the *"Invest Model*". Invest is an acronym of the following words [29]:

- **I** - Independent
- **N** - Negotiable
- **V** - Valuable
- **E** - Estimable
- **S** - Small
- **T** - Testable

According to Bill Wake, a good User Story is supposed to have all these characteristics [29].

Seb Rose analysed the Invest Model rather critically in his talk *"Every Process Needs Thoughtful Participants"* at a conference in 2018. He pointed out that it is not possible for a story to always fulfil all those characteristics. It is not possible for a User Story to always be small and independent since one small User Story is dedicated to achieving a certain goal, a second one might be dependent on that one. Moreover, if this dependency is resolved, an even bigger User Story would be the result. In the end, it might not be possible to fulfil these twos requirements according to the Invest Model. He also criticised the Connextra Template. According to him it is not always necessary to follow a certain form if the main goal of the User Story is fulfilled. On the other hand, he agreed with some basic principles of those methods and stated that the idea of a User Story is to deliver a piece of functionality to achieve the specified goal and a User Story itself is not the requirement [30].

As shown above, there are multiple ways of defining whether a User Story is good or not, how to work with a story or if it has the right size. However, this does not always reflect how a User Story is used in practice. Liskin et al. added that the granularity of a User Story could affect its quality massively. Big User Stories could leave out important information if they are written not clear enough. On the other hand, small stories might not cover all the value the customer wanted to have. Thus, they tried to find a possible measurement of how to define the right granularity of a User Story. Especially big stories were a potential problem, as most of the respondents of their survey stated, that big stories have been too vague or too coarse. A possible way to deal with stories, that are too big is to split them in smaller ones. However, according to their survey, this led to dependencies between the resulting smaller stories, which had to be kept in mind. Splitting a User Story could as well be a powerful practice to discover any gaps of information that must be discussed with the customer to understand the story correctly. In the end, they concluded that there is not a perfect granularity level for a story, as different sizes work better for different stories [31].

In literature there is no definite recipe to get to a perfect set of User Stories. However, there are numerous techniques that show how they can be organised and structured to improve their usage as well as their content.

2.3. Testing

Besides the used software development process models and the techniques used for requirements engineering, testing is another crucial discipline of software development. All development teams have to perform it in order to deliver a successful product.

Back in 2000 James Whitaker claimed in one of his articles, software testing is one of the least understood sections in software development, and almost all bugs are reported due to the following four reasons [32]:

1. Untested code was executed by the real user
2. Executing a process differently in an actual use case than during testing
3. Executing an untested combination of input values
4. Executing the software on an untested environment

He stated that testers must be able to think of any user interaction a piece of software might not be prepared for. Examples might include different operating systems or events like failing power supply or failing network connection. It should be verified that the software reacts properly if any unforeseen event happens. Whitaker advised in his article that one of the most important things a company can do to mitigate such issues is to recognise the complexity of testing and listen to their experts if they warn about degrading quality [32].

According to Rodrigues et al. it has been common that software development and software testing were two completely distinct processes, that were performed by independent teams. However, both teams evaluated the same requirements. This led to discarding almost all testing approaches during the development phase. Even if some testing is partially done automatically with scripts or macros, the testing department and the development team might have performed similar actions. They also added that it is not possible to test all inputs with a Black Box Testing approach. Black Box Testing methods usually are performed without any knowledge of the internal structure of the software under test. This concept is mostly used for validation if the right software is built [33]. Due to the nature of that concept, testing can be very time-consuming. Especially as certain faulty states or errors might only occur after a specific set of operations were performed. This means there might be countless paths which have to be considered during testing. Thus, by only using such a Black Box Testing approach, it is impossible to cover all relevant system paths, regardless of the size of the testing staff. The proposed solution by Rodrigues et al. is to shift more of the testing responsibilities to the development team [34].

Contrary to that, White Box Testing is an approach where the software is tested while knowing the inner structure of it and is usually done by the developers themselves. Its main use case is to detect any logical errors in the code. A typical example of White Box Testing is Unit Testing. These are usually written by developers to test small units of code. A mixture of both approaches, Black Box Testing and White Box Testing, is found at Integration Testing. This type of testing focusses on combining different components of the software to assure that they are working together properly. Typical examples of pure Black Box Testing approaches are Functional Testing and Acceptance Testing. These two types focus on the compliance of the built software with the specified requirements. The difference between these two approaches is that Acceptance Testing is performed by the customer on their environment and Functional Testing is performed by the development team itself. Nevertheless, both are used to test the whole application on a high level. A comparison of all mentioned testing types can be found in Table 2 [33,35].

Type	**Opacity**	**Granularity**	**Scope**
Unit Testing	White Box Testing	Low-Level Design	Small units of code
Integration Testing	White- and Black Box Testing	Low- and High- Level Design	Interactions between different units of code
Functional Testing	Black Box Testing	High-Level Design	Entire Product
Acceptance Testing	Black Box Testing	High-Level Design	Entire Product in customer's environment

TABLE 2: COMPARISON OF DIFFERENT TYPES OF TESTING [33]

2.3.1. Test-Driven Development

Test-Driven Development is an approach highly promoted by Extreme Programming as one of its twelve practices. The basic idea of it is, that before a piece of code is written, automated tests, mostly Unit Tests, are defined. Thus, a set of low-level tests is accumulated over time. With that approach, developers receive instant feedback about the implemented functionality. Furthermore, the automated tests ensure that the functionalities are not broken after another change. Erdogmus et al. challenged this idea with an experiment. As the main result of their study, they were able to show that the participants, who wrote tests before implementing the actual functionality, wrote more tests than the control group, that wrote tests after implementing the functionalities. Another result was that the test-first group ended up being more productive, despite writing more tests. They believe that writing tests before the actual functionality could lead to a better understanding of the problem, which is another advantage of this technique [11,36].

Bhat and Nagappan evaluated this concept in another case study. They stated, that the actual code is only written after the tests. As soon as one of the tests fails, the functionality has to be refactored. The tests provide a low-level design of the software itself as well as reducing newly generated errors by continuously executing tests against the implementation that would highlight any error instantly. Their results indicated that the code quality increased significantly in a software development team using Test-Driven Development in comparison to any other approach inside the same organisation [37].

A similar description of Test-Driven Development was added by Gáspár Nagy and Seb Rose. This description is illustrated in Figure 7. They stated that the development is driven by writing a failing test first. After the functionality is implemented, the newly created test should run successfully. If needed, the code is refactored afterwards. The test cases ensure the correct behaviour of the functionalities even after these changes were made. After that, the next test is written for the next part of the software [38].

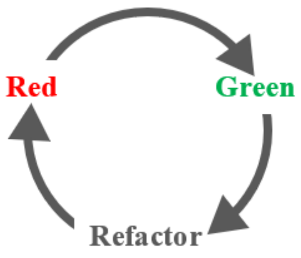


FIGURE 7: SIMPLIFIED TDD-CYCLE [38]

All of that implies that there is one underlying thought that is tried to be accomplished by TDD: not being afraid of changing existing code. This goes back to the Kent Beck, who stated the following in his book *"Test-Driven Development by Example"* [39]:

"Test-driven development (TDD) is a way of managing fear during programming." [39]

2.4. Test Cases as Requirements

The next improvement would be using test cases as requirements. Currently, in most cases, requirement engineering and testing are viewed as two distinctive processes, which are most likely performed by separated teams.

In a case study done by Bjarnason et al. different companies with various approaches on how to use test cases as requirements have been examined using different variants for defining test cases as requirements. One of these companies uses such an approach for the development of Application Programming Interfaces (API). If a new API was created, the new set of requirements should be defined by the system architect. The resulting test suite was used as the most important source of truth during the implementation. This ensured that the built software met the defined requirements. They added that for this use case, namely building APIs, this way of requirements specification represents an efficient way of communication between the development team and the customer. The reason for this efficiency was that only technical roles were involved, and both parties shared a common understanding [40].

Contrary to that, if there are mainly business roles involved on the customer side, the main effort on creating and maintaining the detailed requirements had to be done by the development team. Customers might have an insufficient level of technical understanding to actively participate in the process of generating test cases during this process. They concluded, one of the main challenges in software development projects, keeping up with frequently changing customer requirements, can be overcome by applying practices of defining requirements as test cases. These practices enforce direct and frequent communication between the development team and the customer. Additionally, due to the application of such practices, the created test cases form a living documentation of the application. This documentation can give immediate feedback to the stakeholders at any time throughout the whole life-cycle of the project [40].

Martin and Melnik argue that early writing of Acceptance Tests is a requirements engineering technique and that both interrelate like the two sides of a Möbius strip. They additionally relate to Gause and Weinberg, who said that one of the most effective ways of testing requirements is with test cases like those used for testing the completed system. According to them a Black Box Testing concept could be used during the requirements definition phase because nothing can be opaquer than a box that does not exist yet. That means that those two techniques go hand in hand and cannot be seperated from each other. They came up with the following statement that sums up the relation between tests and requirements [41]:

"In other words, requirements and tests become indistinguishable, so you can specify system behaviour by writing tests and then verify that behaviour by executing the tests." [41]

Thus, Martin and Melnik were able to specify system behaviour by creating tests and later verify the implemented functionality using these tests. To accomplish that, they used FitNesse as their tool of choice for their use case [42]. They concluded, that using tests as requirements can save time and money by focusing on important goals and thereby reducing the creation of unnecessary features and code [41].

Wnuk et al. gave insight into a company that went a different way than what was recommended in literature. They perform very little documentation of their requirements, but they involved Quality Assurance experts very early in the process. The interviewed employees stating that they had no use for long requirement documentations but used the test cases from Quality Assurance as representatives of the requirements. All employees found that the lived process was efficient, and the customer was satisfied with the delivered quality. They stated that several factors might contribute to that success. One of them is open communication in the company. Alongside that, the features form a high level description and focus on the problem and can give feedback through benchmarks and prototypes [43].

2.5. Behaviour Driven Development

All the examples above show that using test cases as requirements, focusing on short communication, high-level problems and early feedback can lead to success. Behaviour Driven Development tries to do exactly that. It combines an agile methodology with requirements engineering techniques that focus on a common understanding with a way on how to automatically check requirements.

As the originator of the term *"Behaviour Driven Development"*, Dan North, stated in his article *"Introducing BDD"*, it evolved out of different agile practices. He added [1]:

"Over time, BDD has grown to encompass the wider picture of agile analysis and automated acceptance testing." [1]

Gojko Adzic stated several key process pattern to such an approach in his work *„Specification by Example"* [2]:

- Deriving the scope from goals
- Specifying collaboratively
- Illustrating using examples
- Refining the specifications
- Automating validation without changing specifications
- Validating frequently
- Evolving a living documentation

This list of key pattern complies with the above-stated description. The living documentation of the system is the result of all those practices. It is a representation of the implemented code that is understandable for all involved people in the form of a domain specific language [2].

Gáspár Nagy and Seb Rose stated in their book *"Discovery"* that Behaviour Driven Development is the missing link between the software and the requirements. According to them, the examples describing the actual functionality keep the business stakeholders involved in the whole process. Those examples are written in natural language, so it is easier to share a common understanding. They added, Behaviour Driven Development itself is an agile practice that is built upon three practices [38]:

1. Discovery
2. Formulation
3. Automation

For Nagy and Rose it is essential to perform all three practices to unleash the full potential of Behaviour Driven Development. Discovery, as the first part, represents a structured approach to mitigate misunderstandings and tries to find a common understanding of a specific goal with examples. With Formulation, the process of documenting the found examples as scenarios is meant, so the business-stakeholders can give detailed feedback instantly. Automation is the last step and is provides the advantage of automatic verification of the specified business-readable specifications from the first two steps to the team. Additionally, the automated tests are providing an up-to-date living documentation of the system. The three steps, defined by Nagy and Rose, are consistent with the key process patterns defined by Gojko Adzic. It could be argued, that the core aspects of such approaches seem to be the same within various experts [38].

In a case study about the usage of test cases as requirements, Bjarnason et al. also investigated Behaviour Driven Development and comparable techniques. They focussed on various shapes of such practices and underlined that breaking down Acceptance Tests to the level of Unit Tests might be too technical for business-stakeholders for understanding the underlying requirements. Therefore, Behaviour Driven Development came into play. As explained earlier, it introduces a domain specific language that tries to prevent misunderstandings between business and technical roles during the implementation process. They came to the conclusion, that direct and frequent communication, which is supported by Behaviour Driven Development, eases the troubles of coping with changes [40].

Furthermore, Unit Tests and Behaviour Driven Development do not exclude each other, but instead, support each other. Usually, Unit Tests drive the development on a low level, as explained earlier in chapter 2.3.1. Behaviour Driven Development wraps around this cycle. This means, a developer starts working on a failing scenario, an Acceptance Test. If this Acceptance Test is failing, the developer adds tests and code, according to Test-Driven Development, until the scenario passes. Afterwards, the produced code is again refactored to ensure that it is organised neatly. After that, the developer proceeds with the next scenario. Acceptance Tests drive the development on a higher level and Unit Tests on a lower level. According to Nagy and Rose, this cycle gives the developers a safety net for the low-level design of the software as well as for the high-level requirements expressed with scenarios.

Moreover, it provides valuable feedback to the business users as the business-readable specifications and the resulting test executions can be read and understood by them. This process, the BDD-Cycle, is illustrated in Figure 8 [38].

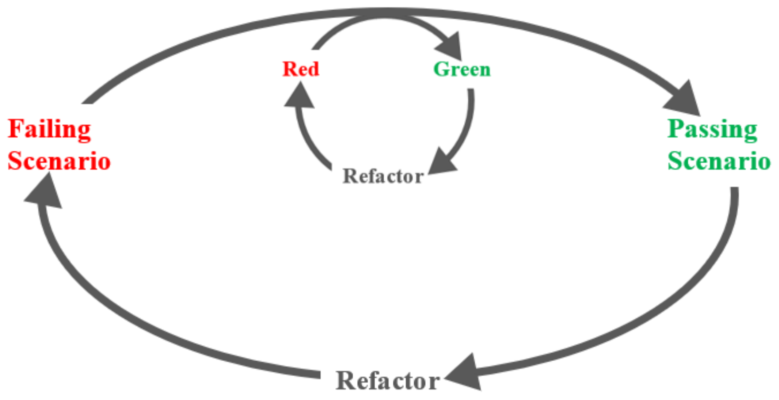


FIGURE 8: BDD-CYCLE [38]

An experience report from Trumler and Paulisch shows the successful application of such an approach in a practical example. They had the task to develop the core components of an MRI (Magnetic Resonance Imaging) scanner software and to ensure that the software had as few errors as possible. According to them the efforts of specifying requirements with examples, formulating these requirements as executable Acceptance Tests and performing Test-Driven Development led to very good results. They were able to deliver the software with a high level of quality and a small number of issues. It was stated that having a big amount of Unit tests as well as to having tests directly derived from the requirements was crucial to avoid reworking of the implemented solution [44].

Behaviour Driven Development can not only be helpful in daily business, but it might also be a possible way to evolve the process of a software development team one step further. In a case study of Olsson et al., the experiences of a company were gathered to serve as input for others who want to move to agile and beyond. The case study focused on challenges in the following areas [17]:

- Adoption of agile methodologies
- Testing practices
- Continuous Deployment
- Customer validation

According to them, some of their challenges can be traced back to the domain-specific nature of their product, an embedded system. The adoption of agile methodologies was a challenge as software and hardware had to be developed, which led to different lengths of development cycles for various parts of the product. Additionally, they stated that dependencies to external partners had been an impediment. For testing practices, the main challenges were maintaining and analysing the automated tests as well as removing old or outdated tests. The most profound challenge for evolving the process towards Continuous Deployment was, to always have an efficient rollback mechanism as that there could be problems with a new version at any time. Regarding customer validation, one big challenge was that the collected data of the customer system was not seen as a potential source of feedback [17].

The challenges from the case study shown above could be mitigated by trying to apply Behaviour Driven Development, as such a process includes techniques that are aimed at solving some of the found problems. As Gáspár Nagy said in his talk at a conference in 2018, Behaviour Driven Development was the way how they could successfully move from Continuous Integration to Continuous Deployment. According to him, the involvement of customer and product management is one of the key aspects to achieve that. As this is enforced by applying such a process a shared ownership of the tests was built up between developers, testers and management. According to him, this was crucial for them to evolve the process. He added that Behaviour Driven Development is not about tests. In their case, it gave them the confidence they needed to move towards Continuous Deployment [45].

It could be said, that Behaviour Driven Development is one way for a software development team to climb another step on the *"Stairway to Heaven"* introduced by Olsson et al., as it could help a team to move from Continuous Integration towards Continuous Deployment [16,17].

As stated above, one crucial step towards that goal is, that the examples can be automated. One way to achieve that is by writing them as scenarios according to the *"Given-When-Then"* template, which was introduced by Chris Matts. This form allows the requirements to be automated and still be readable by humans. There are already tools, such as Cucumber, that enable the team to understand scenarios in that form and hence drive the development through this. Those tools are introduced in section 5.1 in more detail [38,46,47].

However, introducing Behaviour Driven Development can be cumbersome. According to a case study of Gojko Adzic there are several ways on how this methodology can be introduced in a software development team. One of his shown approaches is to implement it as part of a bigger process change. He mentions, that the switch to an agile methodology, like Scrum or Extreme Programming could create an opportunity to introduce the ideas of Behaviour Driven Development as well. For teams, that already have a running process and seek for process improvement, a possible way is the introduction of Functional Testing. If the team does have Functional Tests in place switching to a different tool, capable of executing human-readable specifications, could be one possibility to start the integration. Hence, the introduced methods could be possible first steps to start the implementation of Behaviour Driven Development [2].

3. Environment

The company profiting from the concept of this master's thesis is a specialist for E-Commerce-Platforms. As innovation is one of the core principles of the company, its management always tries to improve the lived processes as well as the quality of the delivered products itself. Both aspects could be improved significantly if Behaviour Driven Development is implemented as part of the software development process of the company. The company itself will be called *"Specialist for E-Commerce-Platforms"*.

On its daily business, the organisation implements E-Commerce-Platforms for a wide range of customer groups. This includes customers with a focus on B2B-platforms, customers with a focus on B2C-platforms as well as mixtures out of those two categories. Hence, the set of requirements that must be dealt with could vary from project to project. As a result of this fact, the software development teams always must cope with new challenges.

As a basis for the majority of the organisation's projects, a basic implementation of an E-Commerce-Platform is used. This standard implementation is available in different peculiarities for different use cases. The development team then adapt this basis to meet the current customers special expectations and needs. With that approach, the Specialist for E-Commerce-Platforms is already able to deliver a working E-Commerce-Platform at an early stage of the project. Throughout the life cycle of the project, this delivered version is changed and extended as required.

The chosen way of working in the organisation of the Specialist for E-Commerce-Platforms is an agile methodology. The primarily chosen one is Scrum. Furthermore, the company is already using a Continuous Integration pipeline. This means the company can be classified as being at the third step of the *"Stairway to Heaven"* as described by Olsson et al in section 2.1.2.4. [16].

The organisation uses User Stories in all its projects to document the needed requirements for the implementation of the desired solutions. In some of the projects the Specialist for E-Commerce-Platforms uses Story Mapping to align on an initial implementation plan. This technique is introduced in chapter 4 [48]. Despite using this technique, none of the requirements are specified in more detail than adding a set of acceptance criteria that must be met for each User Story. This means, that the usage of examples as introduced in chapter 2.5 is not considered so far inside the organisation.

Within the company, various technologies are in use. However, the ones predominately used to implement the functionalities are Java and JavaScript [49,50]. All other technologies are mostly used to aid the development process but do not have a great influence on the actual implementations of specified requirements.

Additionally, the company uses a test framework based on JavaScript, namely WebdriverIO [51]. The usage of this framework implies that the organisation does have knowledge on how such a framework should be used. Moreover, it has already been integrated in the Continuous Integration pipeline of the company. This already used framework is described in chapter 5 in more detail. It must be pointed out, that there have not been any considerations of using a framework capable of executing requirements in the *"Given-When-Then"* format as introduced in chapter 2.5.

As explained earlier, the adoption of Behaviour Driven Development could be a possible step to improve the currently lived process and evolve it to Continuous Deployment. This evolution is one of the current goals of the organisation. Hence, the concept proposed in this thesis is needed as a starting point to accomplish that goal.

4. Requirements Engineering Techniques

As described earlier in chapter 2.5, a concept for the introduction of Behaviour Driven Development must consider various aspects. One of them is the requirements engineering process. There are numerous techniques on how the fastest path to value could be discovered. In this chapter, the following four are described:

1. Feature Injection
2. Story Mapping
3. Example Mapping
4. OOPSI-Model

4.1. Feature Injection

With Feature Injection, Chris Matts introduced a way how a team could focus on the most important part of any piece of software: the value that it brings to the organisation. In an article, he and Gojko Adzic tried to explain the main idea behind this concept [52]. Kent McDonald added a description of these core ideas later. In both works, the same fundamental aspects are listed [53]:

- Identify the value
- Inject the features
- Spot the examples

All the authors mentioned above agree, that if the initial value a piece of software should fulfil is understood by all involved parties the functionalities that might aid the initial goal the most are found. All these defined stories should then be substantiated with examples to spot any special cases, that need to be handled in one way or the other. Additionally, those examples will guide the software development team to think about different possible scenarios. As soon as the initial value is delivered, the cycle starts again, and additional feedback from the already delivered piece of software is available. This feedback should be taken into account for the next iteration [52,53].

This framework is often misunderstood, but, according to McDonald, by following these simple principles, it can be ensured to only build what is relevant [53].

Matts and Adzic added some ideas on how to find the value in their article. One proposal is to ask the involved stakeholders based on an initial request *"why"* they wanted to have that piece of software. This probably has to be repeated a few times until the reason can be identified and used for further discussions [19].

Another proposed way to identify the original reason for a piece of software is to ask the question *"How would that be useful?"*. In his work *"Specification by Example"* Adzic stated, that asking this works better than only asking *"why"* since it starts the discussion without putting someone in a defensive position [2]. Based on the resulting value, the features that may represent the highest initial value should be injected. One important thing to keep in mind is to always start with the outputs. Matts and Adzic added that one of the worst mistakes at any analysis is to start with the inputs. Especially for software requirements, as the inputs do not have any value on their own beside their effect on the outputs [52].

4.2. Story Mapping

In their work *"Fifty Quick Ideas to improve your User Stories"* Gojko Adzic and David Evans present 50 possible solutions to improve on how to work with User Stories. One of those suggestions is the creation of a Story Map [54].

The idea behind Story Mapping came from Jeff Patton originally and is about telling a story of a person, that performs any action to achieve a certain goal. Similar, to the above stated for Feature Injection, it is important to have a goal or a value in mind as a basis for Story Mapping. It is essential to know why the initial request was made. A Story Map differs from a single user story in that it attempts to view the destination as part of the journey the user must take to achieve the original vision. Such a Story Map is built in a workshop. At this workshop it is important that all involved parties are represented. Stakeholders from the business side as well as technical specialists, such as developers and testers, should be present to answer questions and bring up detailed information. The Story Map itself is a grid containing all User Stories needed to fulfil the initial goal. On the horizontal axis the activities and backbone of the story are described. The vertical axis defines what User Story is planned for which release. Thus, all User Stories are placed on the map to determine to which activity they belong to and when they will be delivered. To get to this result, the goal and the User Stories are refined and discussed by all participants of the Story Mapping workshop [48,54,55].

According to Jeff Patton, the process to come up with a Story Map consists of five steps [48]:

1. **Frame**: Defining the purpose the software should fulfil.
2. **Map the Big Picture**: Setting up the backbone of the user's journey, to achieve the goal which was set in the first step.
3. **Explore**: Breaking down the backbone of the Story Map into smaller tasks and discuss all possible ideas.
4. **Slice Out Viable Releases**: Finding a minimum set of all parts of the story that would enable the user to reach the goal in a first release. This first version should be enriched with additional functionalities in the following releases.
5. **Slice Out a Development Strategy**: Planning how the defined releases can be implemented – one after another [48].

A simplified example of such a Story Mapping workshop and the evolution of such a Story Map is shown by Jim Bowes. His chosen domain is an E-Commerce Platform. The main goal of the platform in this example is to enable users to buy a product. The backbone is mapped into three basic features: Product Search, Product Page and Checkout. This is illustrated in Figure 9 [55].

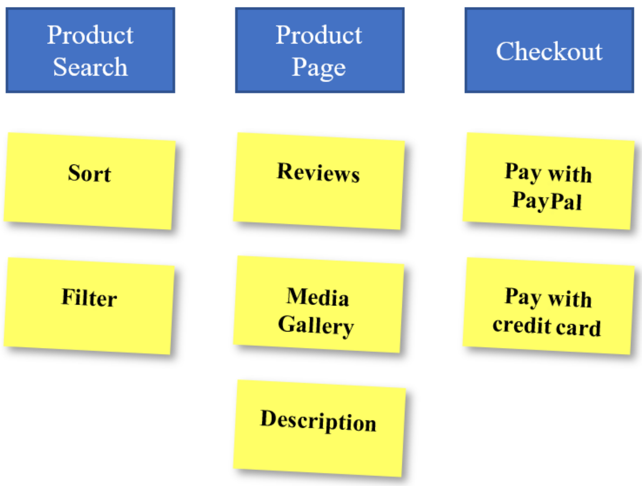


FIGURE 9: STORY MAPPING WORKSHOP – BIG PICTURE [55]

After the parts of the big picture are collected on the map, Bowes added that some further discussions between the team and the stakeholders should show that some of the User Stories should be split into smaller and more detailed parts. Additionally, the position of all User Stories in the defined user journey is set. As pictured in Figure 10, a user must filter the products at first before they can be sorted. After that, the user must select a purchase option before it is possible to add delivery information and payment data. With the integration of time on the axis, it is easier to follow different paths through the process. This eases the process of discussing different possibilities [55].

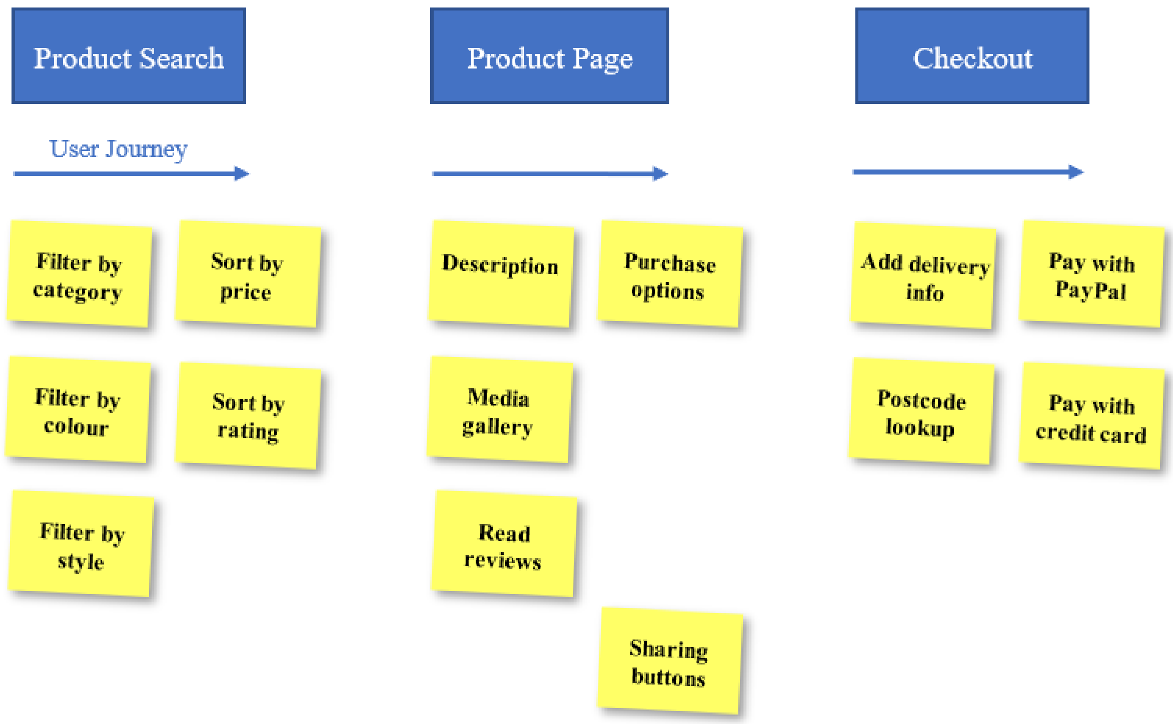


FIGURE 10: STORY MAPPING WORKSHOP – EXPLORED STORIES [55]

According to Bowes, this map makes it easier to find one possible path how a user can buy a product. Filtering the products, showing its description and enabling the user to buy it has the highest priority, as shown in the example above. The team and the business stakeholders agreed that sorting by price or paying with credit card is not important enough to be in the first release. Hence, these functionalities should be done in the second one. Additionally, all stories related to customer reviews are put in the backlog. With this decision, the development team has a clear plan, how they can start delivering the highest possible value in a shorter time, compared to implementing everything in a single release. These decisions are mirrored in Figure 11. With the chosen solution, users are enabled to buy some products without implementing all features of the E-Commerce-Platform. Furthermore, Bowes added, that the process of Story Map itself is not fixed. The story map should be changed as soon as new insights are discovered that might add further wishes or invalidate already taken assumptions [55].

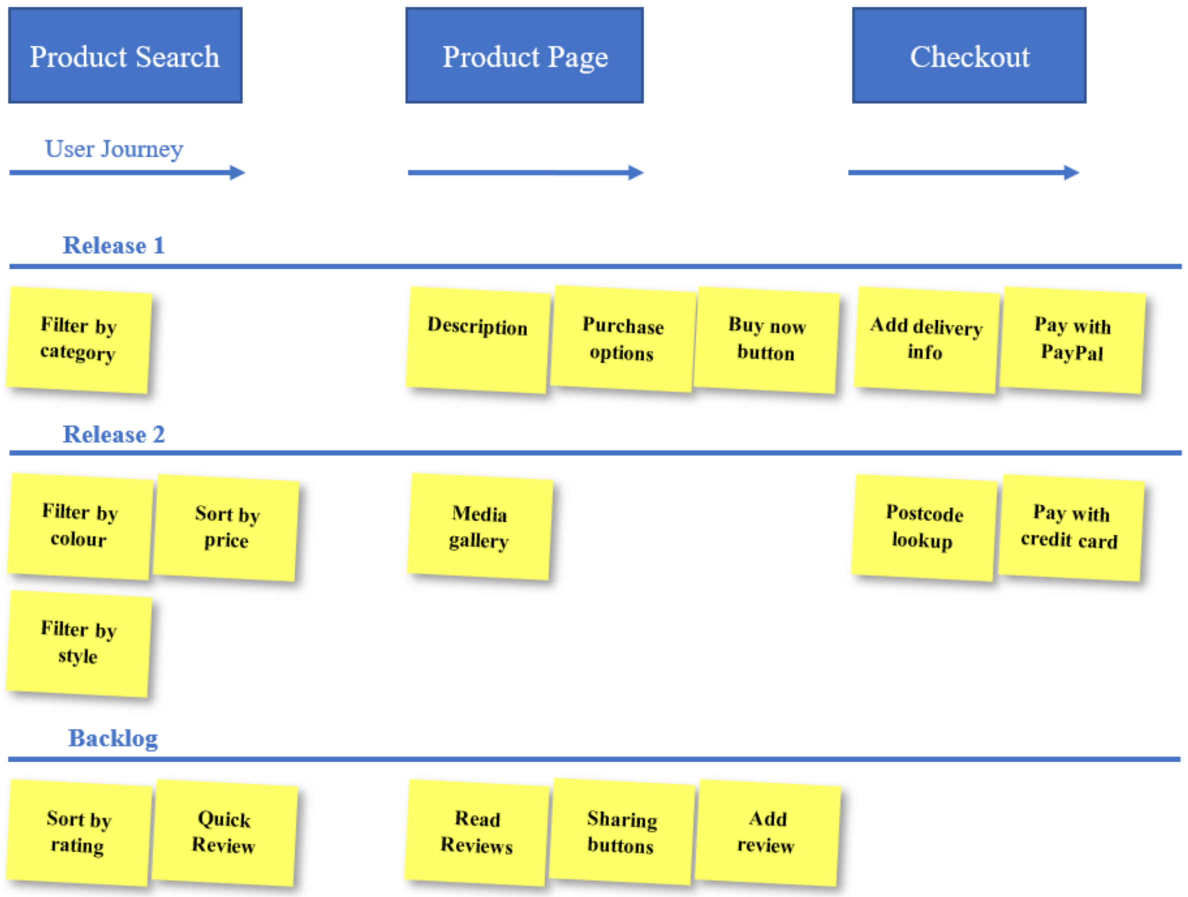


FIGURE 11: STORY MAPPING WORKSHOP – RELEASE SLICES [55]

4.3. Example Mapping

Example Mapping was introduced by Matt Wynne. He described it with the following words:

“I’ve discovered a simple, low-tech method for making this conversation short and powerfully productive. I call it Example Mapping.” [56]

This method is based on four basic elements to structure the conversation at an Example Mapping workshop [56]:

- Stories
- Rules
- Examples
- Questions

Each of these elements is usually represented by a coloured index card. The starting point is a User Story, that shortly describes the goal. User Stories are represented by yellow index cards. Rules are added to the story to specify already known constraints with blue index cards. Each of these rules is then exemplified with one or more examples on green index cards. If during the discussion of the User Story some questions arise, red index cards are used to note down these questions indicating that there is an open point. An abstract example of such a process is shown in Figure 12. According to Matt Wynne, this visual representation will help to find obvious patterns. A lot of red cards indicate that there is not enough knowledge to start the implementation of the User Story. Numerous blue cards signalise a complex User Story that might be necessary to be split into two smaller ones. If many examples are added to a single rule, it might be too complex. This might indicate that there could be a second, currently undiscovered, rule [56].

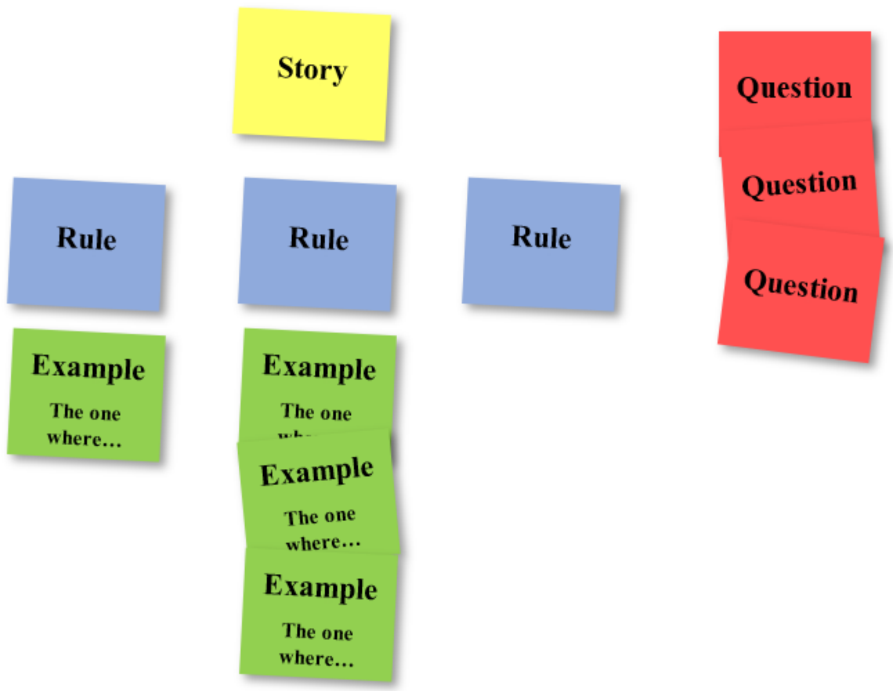


FIGURE 12: EXAMPLE MAPPING OVERVIEW [56]

Matt Wynne stated one way to apply this technique in a workshop is to assemble *“The Three Amigos”*. The Three Amigos are composed out of members from the business-side, developers and testers. From each group, at least one member must participate in the workshop. This does not exclude any other role from the workshop. People with a different background, such as a usability expert, might come up with valuable insights and questions in such a workshop as well. In conclusion, everybody, who can contribute to a specific User Story should participate. However, at least one developer, tester and representative of the business side must be present. Matt Wynne added that it is not necessary during the workshop to formulate the examples in a way that they can be automated. A business user could leave that part to developers and testers and give feedback ones they are done. The focus of this approach is to reach a common understanding of a specific topic between all involved participants [56,57].

4.4. OOPSI-Model

The OOPSI-Model introduced by Jenny Martin, and Pete Buckney is another way to discover requirements as examples in a workshop. They argue that their approach is an extension of Feature Injection, which was already introduced earlier in chapter 4.1. This model aims to find the scenarios with the highest value for business and focus on these first, similar to Story Mapping. For this approach Martin and Buckney suggest, to have at least the Three Amigos present during the workshop, just like for an Example Mapping workshop. The term *“OOPSI”* itself is an acronym out of the following words [58,59]:

- **O** – Outcomes
- **O** – Outputs
- **P** – Processes
- **S** – Scenarios
- **I** – Inputs

The workshop itself is organised to discover the mentioned elements exactly in this order. A possible artefact to start with is a User Story. Such a story can represent the initial goal to be fulfilled. Hence, it is suited to be used as the outcome. An example of that is pictured in Figure 13 [58].

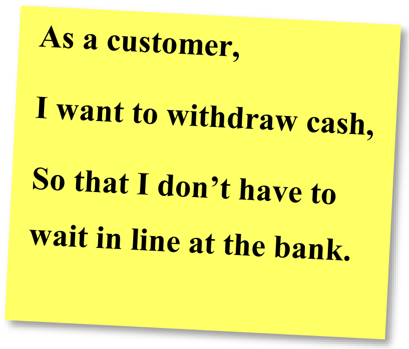


FIGURE 13: OOPSI-MAPPING - USER STORY AS OUTCOME [58]

After the definition of the outcome, the outputs, which are produced to satisfy the initial goal, are discussed. As stated earlier, it is important to have a detailed look at the outputs before the details of the goal are elaborated. For the example stated above, that could be the actual cash, a receipt or any status change like an updated account balance. After a set of outcomes is defined, the most important one is chosen for further discussions. In the shown case, it might be the cash that is given to the customer. According to Jenny Martin, visualising the outputs could aid the discussion. She adds that some of the outputs are inseparable, such as the cash and the updated account balance. The next step of the workshop is to discuss which processes and activities have to be performed to produce the desired outputs. This step allows a discussion of possible paths to the value. Jenny Martin adds that some ideas of other techniques could be applied in this step as well. As an example, Story Mapping can be used to illustrate the user's journey. For the chosen example possible steps are *"Enter Card"*, *"Enter PIN"*, *"Request Cash"* and *"Dispense Cash"* among others. Scenarios are the next step to be discussed after the processes are defined. Usually, a scenario is a possible path through one defined process to generate one of the desired outputs. However, it could also be a simple rule. For her chosen example, it might be sufficient or insufficient funds of the customer or the cash dispenser. Possible results of this workshop after the fourth step are visualised in Figure 14 [28,52,58].

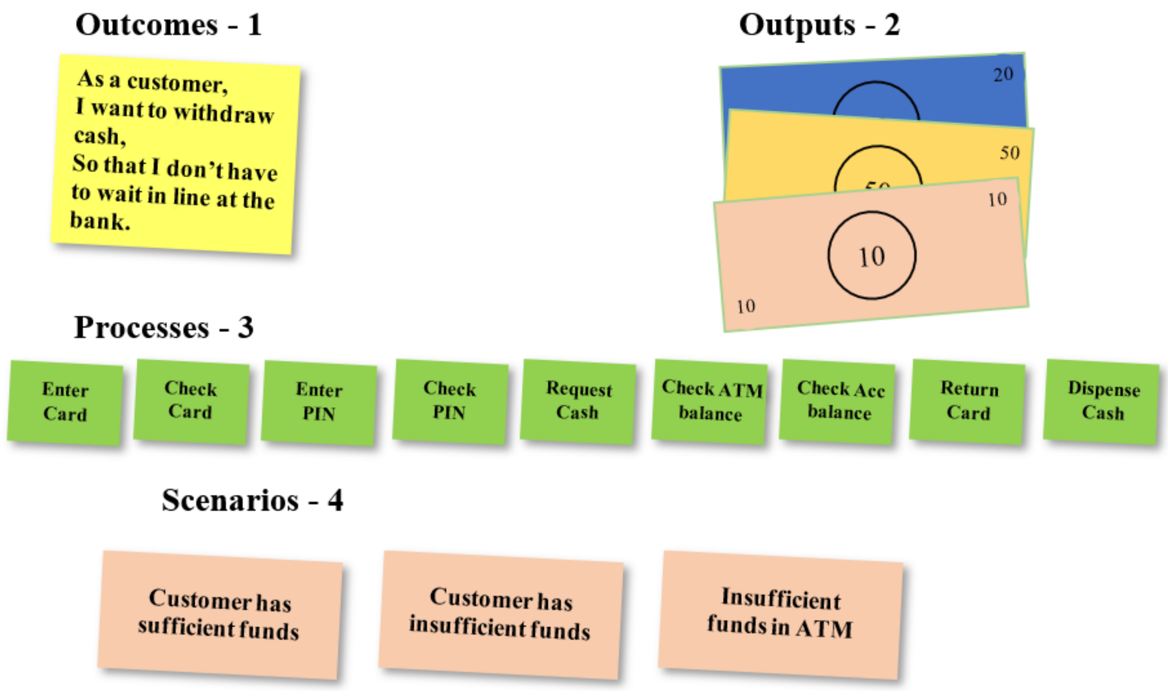

Outcomes - 1
As a customer, I want to withdraw cash, So that I don't have to wait in line at the bank.
Outputs - 2
20
50
10
10
10
Processes - 3
Enter Card
Check Card
Enter PIN
Check PIN
Request Cash
Check ATM balance
Check Acc balance
Return Card
Dispense Cash
Scenarios - 4
Customer has sufficient funds
Customer has insufficient funds
Insufficient funds in ATM

FIGURE 14: OOPSI-MAPPING [58]

Jenny Martin adds the following statement to point out the resulting abstract scenarios are not sufficient for actual development:

"The value of the example is in the data used to drive the example." [58]

Thus, all the discovered scenarios should be filled with actual data, the inputs, to eliminate any existing ambiguities. Therefore, it can be helpful to write the scenarios in the Given-When-Then template, as introduced in chapter 2.5, and add a table to insert different data combinations. It might be possible, that different scenarios could be pictured in one Given-When-Then template and a data table. However, adding different inputs might also indicate, that the found scenarios do not cover all possible cases. Therefore, adding another scenario might be necessary [58].

Another aspect that might aid the discussion in such a workshop is the use of so-called Data Personas. Jenny Martin states that using Data Personas add profiles of actual users to the discussion. Winter et al. agree with Martin by stating that such Data Personas are fictional models of real users that can represent different combinations of data, such as personal information, career information or different skills. Moreover, these personas can be reused across various scenarios. Hence, they can improve the quality of an ongoing conversation.

According to them, this can help the team to envision the real users of the system as persons with a detailed background are represented by such an abstraction [58,60].

Because of the ongoing discussion in an OOPSI-Mapping workshop, Jenny Martin states it should be possible to recognise patterns. Such a pattern can be a grouping of most of the scenarios around different activities of the process stream. Noticeable here is the *“When”* part of the scenarios. It should be possible to set it in relation with a specific process. Moreover, the preconditions, the *“Givens”*, can usually be collated to a process earlier in the stream. Additionally, the postconditions, the *“Thens”*, belong to later processes. These patterns help to find any further ambiguities and to focus on the most important things first. Hence, scenarios that are not crucial for a first release can be addressed in a later iteration [58].

5. Test Automation Tools and Frameworks

To successfully deploy Behaviour Driven Development inside an organisation different tools and frameworks should be used to automate the testing processes. As described earlier in chapter 3, the predominant technologies used by the Specialist for E-Commerce-Platforms are Java and JavaScript. Thus, the investigated tools should be applicable in that environment [49,50]. Due to the fact, that the targeted organisation has some experience with the application of a JavaScript based framework, the focus is set on them.

The process of Behaviour Driven Development, as shown earlier in chapter 2.5, focusses on driving the development on a higher level. This means the focus is set on tools and frameworks enabling a software development team to create automated Acceptance Tests. Thus, the tests should be executed against a running instance of the application. Unit Tests and Integration Tests should not be neglected but do have a different scope than Acceptance Tests. Hence, their application is not discussed in this chapter.

As E-Commerce-Platforms are usually applications that provide a rich Graphical User Interface (GUI) for the customers to interact with, one possible way to write Acceptance Tests is by directly testing the GUI. Emily Bache and Geoffrey Bache show in an experience report about GUI-Testing, that an integral point of such tests in an agile software development project is to support the team throughout the whole development process. According to them, in traditional waterfall projects, that kind of tests was primarily recorded after the part of the GUI has already been implemented. Thus, it has only been possible to spot problems, but not prevent them in the first place. This fact makes recorded tests inapplicable for Behaviour Driven Development. Therefore, a tool must be chosen that allows a team to design the tests before the implementation of the actual functionalities [61].

A common problem to GUI-Tests is the volatile nature of the GUI itself over the life cycle of an agile software project. The application might change several times as new functionalities are added or existing ones are changed. This fact requires a solution that allows the tests to respond to any change of the graphical interface. Emily Bache and Geoffrey Bache state that using page objects is a good solution to this problem. These page objects create a layer of abstraction between the actual test code and the application. Thus, the tests can be designed independently from the changing application. If the GUI changes, only the page objects must be adapted and the tests itself can stay unchanged [61].

The first frameworks in this chapter focus on the execution of human-readable requirements, as explained in section 2.5. Afterwards, some frameworks are introduced, with which a software development team can test the GUI of an application. In this work only open-source solutions are considered.

5.1. BDD Frameworks

In this section two testing tools capable of executing human-readable requirements are introduced. One limitation of this selection is that the tools should be applicable in one of the two mainly used programming languages of the Specialist for E-Commerce-Platforms. Two tools meeting this requirement are Cucumber and Gauge. The most important functionalities of both frameworks for the application in the target environment are introduced in this chapter [47,62].

5.1.1. Cucumber

Cucumber is an open-source project and is currently one of the most used frameworks for Behaviour Driven Development. It is available for most programming languages, not only Java and JavaScript. In the official documentation of Cucumber it is also recommended to use the same language for testing as for the actual implementation [63,64].

The basic concept of Cucumber is to have plain text files, so-called feature files, which hold the definition of the actual requirements. In these files the context, the action and the post-condition are described. These plain text files must be written in a specific grammar – the Gherkin language. These grammar rules define the format in which the requirements have to be written. Beside the feature files, step definitions are needed. These definitions are mapped to the single steps in the feature files and hold the actual test logic. The mapping between the feature files and the step definitions ensures that the specifications of the functionalities stay in a human-readable form. Furthermore, it is also possible to define support code to provide additional functionalities as needed [65,66].

A simplified example how such a feature file could look like is shown in the *"10 Minute Tutorial"* of the cucumber documentation. In this example, as shown in Listing 1, two scenarios are defined for the feature to check if the current day is Friday.

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday
    Given today is Sunday
    When I ask whether it's Friday yet
    Then I should be told "Nope"

  Scenario: Friday is Friday
    Given today is Friday
    When I ask whether it's Friday yet
    Then I should be told "Yes"
```

LISTING 1: GHERKIN EXAMPLE [67]

The step definitions for this feature file are shown in Listing 2. For each of the above defined steps, a step definition must be implemented. The chosen language for the step definitions is JavaScript. It must be mentioned, that in this example the actual functionality under test, the function *"isItFriday"*, should normally be in the actual implementation and not be part of the test code [67].

In Listing 2 it is also illustrated how values can be shared between different steps. The keyword *"this"* is used to store values inside the scope of a scenario execution. This scope is the so-called *"World"* of a scenario and is initialised for each scenario. The clearance of this object is important to not have any data dependencies between different scenarios. For other languages than JavaScript, sharing data is also supported [68].

```
const assert = require('assert');
const { Given, When, Then } = require('cucumber');

function isItFriday(today) {
    if (today === "Friday") {
        return "Yes";
    } else {
        return "Nope";
    }
}

Given('today is Friday', function () {
    this.today = "Friday";
});

Given('today is Sunday', function () {
    this.today = 'Sunday';
});

When('I ask whether it\'s Friday yet', function () {
    this.actualAnswer = isItFriday(this.today);
});

Then('I should be told {string}', function (expectedAnswer) {
    assert.equal(this.actualAnswer, expectedAnswer);
});
```

LISTING 2: STEP DEFINITIONS FOR GHERKIN EXAMPLE IN JAVASCRIPT [67]

The Gherkin language, as already explained, provides grammar rules that enable Cucumber to map test code to the actual requirements. In Listing 1 some of the reserved keywords are already used. The keyword *"Feature"* describes the requirements on a high-level. This description should be short and precise. A *"Scenario"* describes an actual requirement by exemplifying it as a concrete example. Beside the already mentioned keywords, *"Background"* and *"Scenario Outline"* offer some more functionality. The former one is used to define common preconditions for all scenarios in one feature. This is used to avoid duplication of the same preconditions in multiple scenarios. *"Scenario Outline"* on the other hand can be used instead of a *"Scenario"* to run the same scenario multiple times with different test data. This template requires a definition of an *"Examples"* section, that defines the test data for the *"Scenario Outline"* in a table. For each row of the table the template is run once. An example of a Feature that uses *"Scenario Outline"* and *"Background"* is pictured in Listing 3. For all these mentioned keywords, a free-text description can be added to describe the sections in more detail. All mentioned sections use steps to describe the actual functionality. This steps can be used in all of those sections interchangeably [69].

The most commonly used keywords for steps are *"Given"*, *"When"* and *"Then"*, according to the Given-When-Then template. Additionally, the words *"And"* and *"But"* can also be used to indicate the beginning of a step. The usage of the keywords to describe the example as good as possible is recommended, although in Cucumber only the defined text in the step definition matters. This means that the step definition of a defined *"Given"* step in one scenario can be reused as an *"And"* step in another one. Usually *"Given"* is used to set up the context and the preconditions. *"When"* describes the actual functionality under test. A possible example for this is any action that is performed by a user. In the official documentation it is recommended to have exactly one *"When"* in a scenario. This limitation helps to split the scenarios in a way to have an independent definition for each requirement. Having multiple such actions in one scenario might indicate that the design of the specification should be reworked. The keyword *"Then"* is used to check if the action produced the expected output. *"And"* and *"But"* are usually used to extend pre- and postconditions of a scenario [69].

```
Feature: Login
  The Login Feature should enable the user to use the
  functionalities of the account section.

  Background:
    Given The login page is opened

  Scenario Outline: Login as a user
    Given The user enters <username> in the username field
    And The user enters <password> in the password field
    When The user clicks on the login button
    Then The user should see the account section

    Examples:
      | username | password    |
      | "user1"  | "pass1"     |
      | "user2"  | "pass2"     |
```

LISTING 3: GHERKIN – EXAMPLE WITH BACKGROUND AND SCENARIO OUTLINE

Additionally, to the above defined keywords, Gherkin and Cucumber offer other functionalities to organise the test suite. Two of those possibilities are hooks and tags. Tags can be added to Features, scenarios and scenario outlines. Based on that classification, it is possible to only run the specifications with or without a specific combination of tags. Hooks provide the functionality of executing a specific code block before or after each scenario. One possible example for that is opening and closing a browser session if needed by a scenario. Moreover, tags and hooks can be combined as well. Thus, specific setup and teardown functionalities can be applied to scenarios based on their tags. Nevertheless, it is recommended to use hooks only for environmental setup and teardown [70].

Cucumber is a command-line tool. Such a tool can be integrated in any Continuous Integration pipeline as well as on all prevalent platforms. In Listing 4 an example of such a command is shown. This command can take a lot of different arguments specifying the parameters for an execution. It can be specified that the test run should be executed in parallel, among other options. This means that the execution time of a single test run can be reduced [70].

```
$ ./node_modules/.bin/cucumber.js --parallel 5 features/**/*.feature
```

LISTING 4: COMMAND-LINE EXECUTION OF CUCUMBER WITH PARALLELISATION [70]

Trumler and Paulisch used Cucumber successfully as their tool of choice in an experience report. They used it as one of three testing methods in their study. All the techniques they used were defining requirements as examples, Unit Tests for testing the low-level architecture and Cucumber for testing on high level. They tried to reduce the examples in the feature files to the crucial minimum, the so-called *"Happy Path"*, and some additional special cases. With their chosen approach they were able to produce a product with a high level of quality and only a small amount of defects [44].

5.1.2. Gauge

Similar to Cucumber, Gauge is another open-source framework for creating automated acceptance tests. It does not support as many languages as its counterpart Cucumber, but the two most important ones for the Specialist for E-Commerce-Platforms, Java and JavaScript, are included. Although Gauge is not officially a tool for Behaviour Driven Development, as stated by Zabil Maliackal, it offers a lot of functionalities that enables it to be used as one. According to Maliackal, Gauge focusses on testing unlike Cucumber, that tries to improve collaboration [62,71,72].

The biggest difference between Cucumber and Gauge is that the requirements are not defined in Gherkin. Instead of using keywords to indicate the relevant sections of the requirement definitions, Gauge specifications files are written in Markdown. Nevertheless, the underlying concept is similar. Markdown is a well-established markup language. Its main elements are specifications, scenarios and steps. A hashtag *"#"* indicates the start of the specification. Lines starting with two hashtags *"##"* signal the start of a scenario. One specification file can always contain only one specification defined by one or more scenarios. Similar to Cucumber, a scenario is a composition of various steps describing the actual behaviour. The beginning of each step is indicated by an asterisk "*" at the beginning of the line. Using this form can provide human-readable specifications as well. In Listing 5 the same example as for Cucumber is illustrated for Gauge [73,74].

```
# Is it Friday yet?

Everybody wants to know when it's Friday

## Sunday isn't Friday
* today is Sunday
* I ask whether it's Friday yet
* I should be told "Nope"

## Friday is Friday
* today is Friday
* I ask whether it's Friday yet
* I should be told "Yes"
```

LISTING 5: EXAMPLE OF A GAUGE SPECIFICATION [67]

To map the actual test logic to the scenarios, Gauge uses step definitions. For each line starting with an asterisk, a step definition must be defined. Such a step definition is illustrated in Listing 6. In this example, JavaScript is used to demonstrate the implementation of the step definitions [74].

As sharing data between step definitions is necessary, it must be possible to do so. Gauge supports this functionality by providing three different data stores, all having a different scope:

- ScenarioStore
- SpecStore
- SuiteStore

Each of the mentioned data stores is cleared based on the defined scope. The ScenarioStore is cleared after the execution of every scenario, the SpecStore is only available in the life cycle of one specification and the SuiteStore can be used throughout the whole execution of the test suite [74].

```
const assert = require('assert');

function isItFriday(today) {
    if (today === "Friday") {
        return "Yes";
    } else {
        return "Nope";
    }
}

step("today is Friday", async function() {
    let today = "Friday";
    gauge.dataStore.scenarioStore.put('day', today);
});

step("today is Sunday", async function() {
    let today = "Sunday";
    gauge.dataStore.scenarioStore.put('day', today);
});

step("I ask whether it's Friday yet", async function() {
    let answer = isItFriday(gauge.dataStore.scenarioStore.get('day'));
    gauge.dataStore.scenarioStore.put('answer', answer);
});

step("I should be told <answer>", async function(answer) {
    assert.equal(gauge.dataStore.scenarioStore.get('answer'), answer);
});
```

LISTING 6: STEP DEFINITIONS FOR GAUGE EXAMPLE IN JAVASCRIPT [67]

Additionally, to the already mentioned basic functionalities of the framework, it is also possible to define steps for the setup and teardown of a scenario. The setup section is called *"Context"* and consists out of one or multiple steps. These steps have to be defined above the first scenario. The context steps are then executed before each of the defined scenarios. Their counterpart is the teardown section, which is indicated by a minimum of three underlines *"___"* at the beginning of a line. This section must be defined underneath all scenarios in the specification file. All defined steps in the teardown section are executed after each scenario. Another feature of Gauge is the usage of data tables to execute a defined scenario multiple times based on the values in the table. Listing 7 illustrates the usage of the just mentioned functionalities of the framework [74].

```
# Login
  The Login Feature should enable the user to use the
  functionalities of the account section.

      | username | password     |
      |----------|-------------|
      | "user1"  | "pass1"      |
      | "user2"  | "pass2"      |

* Open the browser
* The login page is opened

## Login as a user
* The user enters <username> in the username field
* The user enters <password> in the password field
* The user clicks on the login button
* The user should see the account section

___
* Close the browser
```

LISTING 7: GAUGE - EXAMPLE WITH DATA TABLE, CONTEXT AND TEARDOWN

Furthermore, Gauge also provides the functionalities of hooks and tags. Tags can be specified on specification and scenario level. They can be used to organise the test suite as well as to execute only specific specifications and scenarios. Hooks can be used to execute specific test logic before or after the following scopes:

- Suite
- Specification
- Scenario
- Step

These hooks can as well be executed based on specific tags, which gives further flexibility at the organisation and execution of the test suite [74].

Additionally, Gauge allows to define so-called *"Concepts"*. This functionality combines different steps under one definition. Such concepts can be defined in separated files with the ending *".cpt"*. Each concept is a combination of normal steps. All the defined concepts can be used like any other step inside the specification of a scenario. This functionality allows to combine multiple steps into a single batch to make the specifications more readable. Furthermore, it helps to avoid the duplication of step definitions [74].

To make the setup of a Gauge test suite easier, it provides a set of templates to initialise its basic structure. As Gauge is a command line tool, it can as well be started with different parameters, such as parallel execution. In Listing 8 commands for initialisation and execution of a Gauge test suite are illustrated [75].

```
# initialise JavaScript project
gauge init js

# parallel execution of all specifications inside the spec-folder
gauge run --parallel -n=5 specs/
```

LISTING 8: INITIALISATION AND EXECUTION OF GAUGE TEST SUITE

5.2. GUI-Testing Frameworks

In this chapter some frameworks are introduced that enable a development team to write tests simulating a user’s behaviour in a web browser. An important limitation of considering such tools for this thesis is that the tools can be used in conjunction with one of the frameworks introduced in chapter 5.1. There are numerous tools capable of doing so and the following ones are explained in detail:

1. Selenium WebDriver [76]
2. WebdriverIO [51]
3. Puppeteer [77]

5.2.1. Selenium WebDriver

According to their documentation, Selenium is probably the most widely used framework for automating test cases directly in browsers. The first version of it was developed in 2004 and has evolved ever since. Throughout the life cycle of the Selenium project, various versions and sub-projects have been developed. For the use case of the Specialist for E-Commerce-Platforms, the current version of Selenium WebDriver is a conceivable option. Furthermore, Java and JavaScript are among the supported programming languages for this framework [76,78].

The main goal of Selenium WebDriver is to have a well-structured framework to support testing of dynamic web applications. It uses the WebDriver API to communicate with the browser. This WebDriver API provides an interface to control the behaviour of a browser. Its protocol is independent of any language and browser, which means this abstraction is applicable for all of them. However, each browser, that should be automated using Selenium WebDriver, must provide its own implementation of that protocol. These implementations are called *"drivers"*. Each of these drivers can therefore be used to execute the same operations on a different browser. This makes Selenium WebDriver capable of running the same set of tests on different browsers by just exchanging the used driver, making it possible to ensure that the same functionality works across various environments. For executing the test suite on a local environment only one such driver and the corresponding browser is needed. Additionally, a Selenium Server can be used to manage the execution for different drivers [79,80].

If the test suite should be executed on multiple browsers or specific browser versions, a Selenium Grid, another Selenium project, can be used. It allows to run the tests on different browsers as well as to execute them in parallel. The Selenium Grid manages the distribution of the tests and hereby provides a scalable architecture for managing test executions [81].

Selenium WebDriver itself does not come with a test runner, which means that it can be integrated in any test runner available for the supported languages. This provides a high level of flexibility and enables it to be possibly integrated with either Cucumber or Gauge.

5.2.2. WebdriverIO

WebdriverIO is a JavaScript testing framework that provides a simple interface for writing automated tests. Similar to Selenium WebDriver, WebDriverIO implements the WebDriver API and can use the same drivers for different browsers to simulate a user's behaviour. It includes different services that provide various functionalities. One of these services is the selenium-standalone service, that provides support for the different browsers. This service builds up the needed Selenium Server to interact with all required drivers. This test framework also supports the automation of tests for mobile devices by implementing the Appium Protocol. This integration is done as well via a service, which means it can be included in the configuration of the framework [51,82,83].

Besides all services that enable integrations to various other utilities, WebdriverIO uses a predefined set of frameworks as test runners. Among those frameworks is Cucumber, one of the already in section 5.1 introduced BDD frameworks [84].

5.2.3. Puppeteer

Another framework capable of automating acceptance tests in a browser is Puppeteer. Like WebdriverIO it is a JavaScript framework. Other than the two already introduced frameworks, Puppeteer does not implement the WebDriver API to control a browser. It implements the Chrome DevTools Protocol [85]. A result of that is that only Chromium-based browsers are supported [86]. Nevertheless, there are already efforts to provide support for other browsers, such as Firefox, by implementing this protocol as well [77,87].

The Puppeteer project is mainly maintained by the Chrome DevTools Team, which means directly by Google, despite being an open-source framework. Hence, everybody can contribute to the project. It is based on the same principles as the Chromium project [77,88]:

- Speed
- Security
- Stability
- Simplicity

As one of their main goals Puppeteer aims at providing a simple library to use the DevTools Protocol. This reference implementation could, according to the official documentation, as well be used as a foundation layer for other frameworks. Another goal is to put more emphasis on the growth of automated headless testing. Headless testing means, that the operations can be executed without having a visible representation of the browser. Thus, all defined operations will be executed in a background process. This makes the execution fast in comparison to a non-headless setup. Puppeteer runs headless by default but also provides the functionality to change this setting [77].

In the official documentation it is also added that Puppeteer is not intended to be a replacement for Selenium. It focusses on providing more functionalities and a higher reliability. Whereas Selenium aims at being compatible with a lot of different browsers. Hence, its focus is set on cross-browser test automation [77].

Puppeteer does not provide any framework to execute a test suite. It is built in a way, that it can be integrated in any JavaScript application. This makes it possible to integrate it with any of the shown frameworks in chapter 5.1.

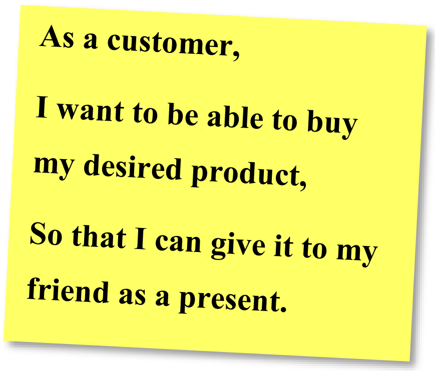
6. Proposed Concept

The proposed concept is based on already introduced techniques and frameworks. It should help a software development team to discover their requirements more easily as well as to aid their continuous verification. Therefore, a technique for the requirements engineering process is suggested. Furthermore, a proposal which tools and frameworks could be used within the given environment is added. After that, a recommendation is given how the two parts of this concept can be introduced in the development process of the Specialist for E-Commerce-Platforms.

Additionally, this concept tries to help the Specialist for E-Commerce-Platforms to climb another step of the *"Stairway to Heaven"*, as introduced in chapter 2.1.2.4. Once the development team has built up enough confidence in the code base it has produced, it can be considered to deploy any change directly to a pre-production system, if all scenarios have been executed successfully on the test environment. This can be extended even further, by deploying directly on a production system. As already stated earlier in section 2.5 by Gáspár Nagy, confidence in the code base and its verification is the key element to make the step from Continuous Integration to Continuous Deployment [45]. The proposed concept aims at giving a development team the needed guidance and all required tools to raise its confidence to do exactly that.

6.1. Requirements Engineering

One crucial thing about requirements engineering is to understand why a specific functionality is needed. As shown by Mike Cohn, User Stories encourage verbal communication to limit misunderstandings. Thus, User Stories build a solid ground to build upon for working with an agile software development process model [24]. Therefore, a User Story is used to illustrate an example as a starting point of this concept. This story can be seen in Figure 15.


As a customer,

I want to be able to buy my desired product,

So that I can give it to my friend as a present.

FIGURE 15: GOAL AS A USER STORY

With the User Story as a starting point, the goal can be examined in more detail. In chapter 4, some possible ways to do this have already been introduced, namely Story Mapping, Feature Injection, Example Mapping and the OOPSI-Model. Instead of using one of these techniques, another one is proposed: the *"Extended OOPSI-Model"*.

6.1.1. Extended OOPSI-Model

This approach is a slight adaption of the OOPSI-Model from Jenny Martin and Pete Buckney, which has already been introduced earlier in chapter 4.4 [58]. It provides higher flexibility and more structure than the original OOPSI-Model.

6.1.1.1. Extended OOPSI-Model: Concept

The main idea behind this model is to split up the discussions during a workshop into smaller chunks. Especially, if the complexity is too high to discuss a big goal at once. The starting point for the Extended OOPSI-Model is the same as for the regular OOPSI-Model. As shown earlier in chapter 4.4, following the OOPSI-Model may reveal various patterns. If those patterns suggest, that there is a high level of complexity that can be assigned to only one part of the processes, this step should be divided into various others. However, this means as well, that there can be multiple additional process steps which could be added. Instead of just adding new process steps to the initial ones, starting another OOPSI-Model is proposed to keep the big picture as simple as possible. Both, the parent and the child model, are linked through the order of the process chain of the parent model. This means that the resulting scenarios from the child process can be used as pre-conditions for the scenarios of the following process steps of the parent.

As a starting point for the child OOPSI-Model, a goal has to be defined to represent the initial step in the process chain. This could be done by formulating a User Story dedicated to this part of the process. Thus, the initial discussion on the higher level can continue to identify the highest value first. It is also possible to concentrate on the details of that process step next but there could be other steps in the process chain that should be split as well.

After the discussions are over, there should be a set of examples that enables the development team to deliver a first, most likely minimal, version to fulfil the initial goal. This approach brings one of the oldest principles of software development, and other domains as well, to the OOPSI-Model: **Divide & Conquer**.

After explaining the main idea of this concept, an example of the application of this model is shown in the next section. Initially, the procedure will be the same as for the original OOPSI-Model, but as the example advances, the advantages of the adaption to this model will be pointed out. The example itself is demonstrated as an ongoing workshop in which the goal is discussed in more detail.

6.1.1.2. Extended OOPSI-Model: An Example

The first step, the outcome, is defined in Figure 15, hence the next one is to think of possible outputs, that are produced during the completion of the goal. For the chosen example, these could be various artefacts, data sets or a changed status, just as the OOPSI-Model suggests. Some of the possible outputs for the chosen case are listed below:

- Address data of the user
- Payment data of the user
- Consent to Terms and Conditions
- Order object
- Order confirmation mail
- Updated stock-level of the product

For the discussion, it can be helpful to visualise these outputs. Hence, it is advised to draw visual representations instead of just writing words on a list. After no participant can add any further outputs to the list, the next step is to discuss which of these outputs is the most important one. Some participants might say, that all found outputs are equally important in order to fulfil the goal. The address data is needed to know where the product should be shipped to and the consent to the terms and conditions has to be given for legal reasons. However, all agree that the order is the object that must contain all relevant information. Thus, it can be assumed that it is the most important output.

The next thing to be discussed are the process steps needed to reach the goal. As already pointed out earlier in chapter 4.4, Jenny Martin suggested that it is advisable to include other techniques, such as Story Mapping by Jeff Patton, to drive the conversation [58]. The idea is to tell a story about how the customer will be able to buy the product step by step. Looking back to the chosen example, the customer wants to buy a present for his friend. To make the discussion more substantial, a name is given to the customer as well as for the friend. This so-called Data Personas might help the workshop participants to follow the conversation more easily [60]. Thus, the discussion focusses on the story of how John finds and buys a present for his friend Peter. In Figure 16 an example of the process steps is illustrated.

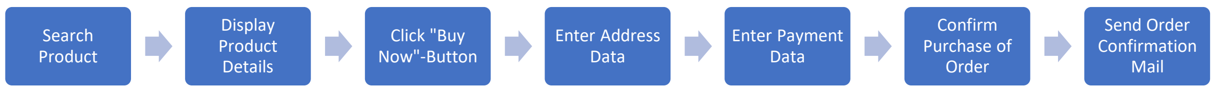


FIGURE 16: PROCESS STEPS

After looking back to the defined outputs, it is obvious that some of the outputs can be directly assigned to some of the found process steps. However, the output *"Updated stock-level of the product"* cannot be related with any of them. Hence, there probably is another step in this chain, which has not been considered so far. This suggests that it is always good advice to recheck with any of the previous higher-level steps of the OOPSI-Model to avoid disregarding any important points during the discussion.

The next step in the OOPSI-Model is to take the most important step of the process chain and refine it in more detail with scenarios. As the order object was defined as the most important output, the process steps that can be related to it should be identified. The one fitting best is probably *"Confirm Purchase of Order"*. This also aligns with Story Mapping, as both techniques have a common goal – refining the stories in more detail and making the user journey understandable. Only the used artefacts of the two methods differ.

One of the first scenarios that comes up represents the *"Happy Path"*. This scenario is the most basic path that must work. It could be called *"Purchase of order successfully"*. Additionally, another scenario is discussed to specify what would happen if John provided an incorrect address. If these scenarios are now rephrased in Gherkin, as introduced in chapter 5.1.1, it may look like shown in Listing 9.

```
Feature: Order a Product

  Scenario: Purchase of Order successfully
    Given John chose a product
    And John provided a correct address
    And John provided correct payment data
    And John approved to the terms and conditions
    When John clicks to confirm the purchase of the order
    Then the order object should be created
    And a order confirmation mail should be sent to John's email
    And the stock level of the product should be updated

  Scenario: Wrong address data
    Given John chose a product
    And John provided an incorrect address
    And John provided correct payment data
    And John approved to the terms and conditions
    When John clicks to confirm the purchase of the order
    Then the order object should not be created
    And John should see a message about the incorrect address
```

LISTING 9: PROPOSED CONCEPT: ORDER SCENARIOS

These scenarios are discussed again in the workshop, and several questions come up from different participants:

- How do we know John's e-mail address? We currently only have his address data and payment data.
- What is an incorrect address?
- There is an error message. Why haven't we thought about that while discussing potential outputs?
- If we show error messages, shouldn't we show them right after the user entered incorrect data and prevent any further advancing in the process?

By only discussing the first two scenarios, the workshop participants might have a lot of questions, which suggest, that the complexity is higher than initially expected. Probably, there should be a step added to enter personal information, such as John's e-mail address.

Additionally, another step taking care of the validation of personal and address data should be added. Within this step the definition what a correct address is and how it must be structured might be addressed. This would be even more obvious if the scenarios would have already been undermined with concrete inputs. However, in the shown case the scenarios give enough insight. Hence, it has not been necessary to proceed with the last step for now. Moreover, by taking a closer look at the provided scenarios, it can be argued, that there is a pattern in them. As already explained at the introduction of the OOPSI-Model, some outputs can be assigned to specific steps in the process. In the presented example, address data, and probably personal data, can be assigned exactly to the same step in the process: *"Enter Address Data"*. The result of this step acts as a precondition for the subsequent step *"Confirm Purchase of Order"*.

In such a use case the Extended OOPSI-Model can show its strength. Instead of just adding, and thereby extending the process chain by more and more steps, another OOPSI-Model can be started. The outcome of this child model is dedicated to a step in the process chain. In the shown example, the step *"Enter Address Data"* can be changed to *"Enter Personal and Address Data"* and another OOPSI-Model is started. Therefore, a goal must be defined as a starting point, that defines what has to be achieved in the child model. This could again be a User Story, which might look as shown in Figure 17.

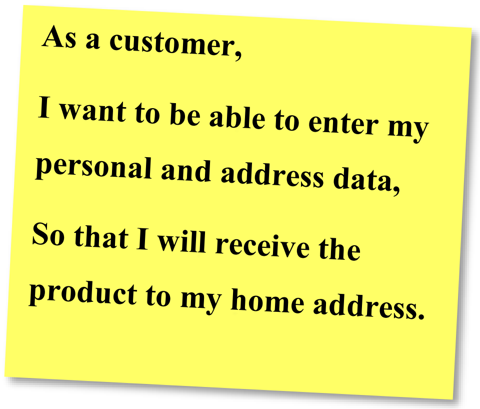

As a customer,
I want to be able to enter my personal and address data,
So that I will receive the product to my home address.

FIGURE 17: USER STORY FOR ENTERING ADDRESS DATA

The workshop participants might decide to refine this goal right away, if they think it is important enough. It would as well be possible to proceed with the high-level process chain in that situation, as the goal itself is specified and therefore is ready to be discussed later.

However, in this example the workshop continues with the child model. As the goal is defined, the first things to be clarified are the outputs, which are exemplified in the following list:

- E-mail of the customer
- Name of the customer
- Postal code of the customer
- Street name and street number of the customer
- Error messages

After the participants are sure all possible outputs are covered, they start with the next step, the processes, for this child model. As it is a child process of the initial goal, the resulting process chain can be much shorter than the initial one, as all the other steps are already provided. These processes represent a detailed specification of the parent's process step and is illustrated in Figure 18.

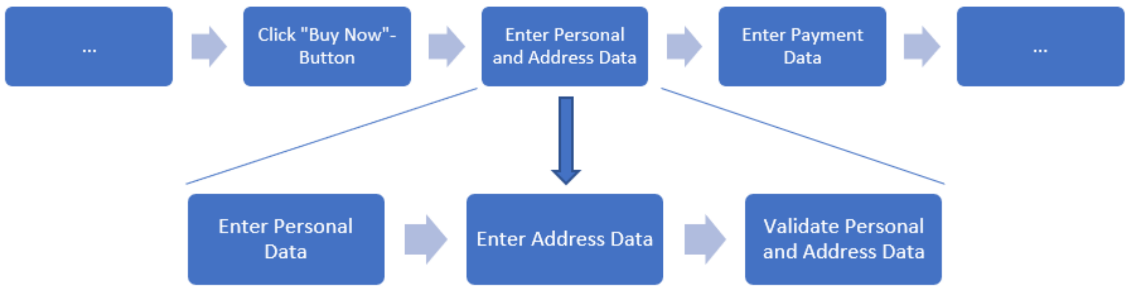


FIGURE 18: EXTENDED OOPSI-MODEL – PROCESSES EXAMPLE

After the process chain is represented as well, the details can be discussed again. Therefore, some scenarios are needed. The first one will again represent the *"Happy Path"*. Some others are added to highlight that there can be invalid data combinations. The first iteration of the scenarios is shown in Listing 10.

```
Feature: Enter Personal and Address Data

  Scenario: Correct Personal and Address Data provided
    Given John chose a product
    And John sets a correct name
    And John sets a correct email
    And John sets a correct street
    And John sets a correct postalcode
    And John sets a correct city
    When John clicks to confirm his provided information
    Then John should be forwarded to the Payment Page

  Scenario: Incorrect Personal and Address Data provided
    Given John chose a product
    And John sets a correct name
    And John sets an incorrect email
    And John sets an incorrect street
    And John sets a correct postalcode
    And John sets a correct city
    When John clicks to confirm his provided information
    Then an error message should be shown
```

LISTING 10: EXTENDED OOPSI-MODEL – SCENARIOS FOR DATA VALIDATION

From these scenarios, it is possible to derive the information that they can be assigned to the step *"Validate Personal and Address Data"* in the process chain because the action, the *"When"*, represents the validation of the data John provided. The preconditions for this scenario are providing the personal and address data as well as choosing a product beforehand, which is represented by the process of the parent OOPSI-Model. Similar to that, the postconditions represent some error messages if the validation failed as well as the next step of the original process chain. In this example, John would progress to the point in the process, where he could provide his payment information if the validation succeeded. Furthermore, the scenarios show that both are quite similar in their structure. This is important for the next step, providing real data for the scenarios. In Listing 11 an example is shown, which enriches the scenarios with real inputs.

```
Feature: Enter Personal and Address Data

  Scenario: Correct Personal and Address Data provided
    Given John chose a product
    And John sets name to be "John Doe"
    And John sets email to be "john@example.com"
    And John sets street to be "Herrengasse 1"
    And John sets postalcode to be "1010"
    And John sets city to be "Wien"
    When John clicks to confirm his provided information
    Then John should proceed to the Payment Page

  Scenario: Incorrect Personal and Address Data provided
    Given John chose a product
    And John sets name to be "John Doe"
    And John sets email to be "@example."
    And John sets street to be ""
    And John sets postalcode to be "1010"
    And John sets city to be "Wien"
    When John clicks to confirm his provided information
    Then the message "Incorrect Data entered" should be shown
```

LISTING 11: EXTENDED OOPSI-MODEL – INPUTS FOR PERSONAL AND ADDRESS DATA

Although these scenarios provide much more insight into the correct or incorrect data set for personal and address data it is still not sufficient to start development. This can be improved by giving more real inputs for a better understanding. By using some features of the Gherkin language, Backgrounds and Scenario Outlines, it is possible to group different examples together and add test data in a table, so the scenario is easier to read [89]. Some of the examples are:

- Name is too short
- E-mail address has a wrong format
- Name of the city contains characters that are not allowed

All these examples are added to Scenario Outlines, as illustrated in Figure 12.

```
Feature: Enter Personal and Address Data

  Background:
    Given John chose a product

  Scenario Outline: Correct Personal and Address Data provided
    Given John sets name to be <name>
    And John sets email to be <email>
    And John sets street to be <street>
    And John sets postalcode to be <postal>
    And John sets city to be <city>
    When John clicks to confirm his provided information
    Then John should proceed to the Payment Page
    Examples:
      | name          | email                       | street          | postal | city   |
      | "John Doe"    | "john@example.com”          | "Herrengasse 1" | "1010" | "Wien" |
      | "Peter Test"  | "peter.test@example.com”    | "Test 3"        | "8010" | "Graz" |

  Scenario Outline: Incorrect Personal and Address Data provided
    Given John sets name to be <name>
    And John sets email to be <email>
    And John sets street to be <street>
    And John sets postalcode to be <postal>
    And John sets city to be <city>
    When John clicks to confirm his provided information
    Then the message "Incorrect Data entered" should be shown
    Examples:
      | name        | email              | street          | postal      | city   |
      | "JD"        | "john@example.com” | "Herrengasse 1" | "1010"      | "Wien" |
      | "John Doe"  | "@example.”        | "Herrengasse 1" | "1010"      | "Wien" |
      | "John Doe"  | "john@example.com” | ""              | "1010"      | "Wien" |
      | "John Doe"  | "john@example.com” | "Herrengasse 1" | "10101010"  | "Wien" |
      | "John Doe"  | "john@example.com” | "Herrengasse 1" | "1010"      | "$$##" |
```

LISTING 12: EXTENDED OOPSI-MODEL – SCENARIO OUTLINES

After this discussion one of the workshop participant asks, why John should not be able to send the present directly to Peter. In such a case, John should have the possibility to provide a second address to distinguish between delivery address and billing address. This is a perfect example for an additional goal that could be implemented after the most crucial set of functionalities is delivered. Hence, this goal can be written down as a User Story, which will be a starting point for another workshop in the future.

At this stage, there are some examples with test data determining how the application should behave for different sets of personal and address data. This enables the discussion to focus on the originally started OOPSI-Model. In Listing 9 two scenarios have been stated, one for a successful order process and another one for a non-successful order process. In the chosen example, John must provide correct personal and address data to be able to advance to the next step in the process. This is enforced by the scenarios, that have been defined in the child model. Thus, it is not necessary to recheck the scenario with invalid personal or address data. That leads to the next advantage of the Extended OOPSI-Model. Through splitting up the process chain of the original OOPSI-Model, it is possible to examine parts of the goal in detail and provide the results as preconditions for the subsequent steps of the user’s journey. Although the resulting scenarios are depending on each other, they are only dedicated to exactly one behaviour. This will help to keep the resulting requirements organised. The result could look like shown in Listing 13.

```
Feature: Enter Personal and Address Data

  Background:
    Given John chose a product

  Scenario Outline: Correct Personal and Address Data provided
    Given John sets name to be <name>
    And John sets email to be <email>
    And John sets street to be <street>
    And John sets postalcode to be <postal>
    And John sets city to be <city>
    When John clicks to confirm his provided information
    Then John should proceed to the Payment Page
    Examples:
      | name          | email                     | street            | postal | city   |
      | "John Doe"    | "john@example.com         | "Herrengasse 1"   | "1010" | "Wien" |
      | "Peter Test"  | "peter.test@example.com   | "Test 3"          | "8010" | "Graz" |

  Scenario Outline: Incorrect Personal and Address Data provided
    Given John sets name to be <name>
    And John sets email to be <email>
    And John sets street to be <street>
    And John sets postalcode to be <postal>
    And John sets city to be <city>
    When John clicks to confirm his provided information
    Then the message "Incorrect Data entered" should be shown
    Examples:
      | name        | email              | street           | postal       | city   |
      | "JD"        | "john@example.com  | "Herrengasse 1"  | "1010"       | "Wien" |
      | "John Doe"  | "@example.         | "Herrengasse 1"  | "1010"       | "Wien" |
      | "John Doe"  | "john@example.com  | ""               | "1010"       | "Wien" |
      | "John Doe"  | "john@example.com  | "Herrengasse 1"  | "10101010"   | "Wien" |
      | "John Doe"  | "john@example.com  | "Herrengasse 1"  | "1010"       | "$$##" |

Feature: Order a Product

  Scenario: Purchase of Order successfully
    Given John chose a product
    And John provided correct personal and address data
    And John provided correct payment data
    And John approved to the terms and conditions
    When John clicks to confirm the purchase of the order
    Then the order object should be created
    And an order confirmation mail should be sent to John's email
    And the stock level of the product should be updated
```

LISTING 13: EXTENDED OOPSI-MODEL – RESULTS AFTER CHILD PROCESS

After that, the discussions should continue with adding more and more scenarios to the initial process chain, such as adding valid and invalid payment data. This is possibly another scenario, where it might be advisable to split the discussions into smaller parts to keep the big picture as clear as possible. The resulting scenarios of the workshop provide an unambiguous set of requirements for the development team.

During development or any following workshop, some question might arise about potential scenarios, that have not been considered. However, if such scenarios come up, the Extended OOPSI-Model will give guidance where the scenario could fit in and how the team can find a possible solution for it. Moreover, the resulting scenarios can be used for the automation of test cases. If they are already written in Gherkin language during the meeting, they can be integrated right away to the test suite and drive the development as suggested in chapter 2.5. Not writing the scenarios in Gherkin means additional effort after the workshop as the scenarios must be formulated to be automated. Nevertheless, this could speed up the workshop itself, as already pointed out by Matt Wynne for Example Mapping [56]. It should be tested and checked which approach is preferred by the development team, as the workshop aims primarily at finding a common understanding of the goal.

Before the development team can start with the actual implementation, it might be advisable to structure the upcoming work in reasonable work packages. The reasons to do this are numerous. One reason could be that in some agile software development process models, such as Scrum, estimations should be made in order to know if a backlog item can be implemented in one iteration. Estimating the single process steps could be a possible way. Alternatively, it could be a good idea to group related scenarios and form a work package out of them. Which variant the team choses is up to the complexity of the scenarios itself. It could be that one scenario might keep a developer busy for a whole iteration and on the other hand, for less complex ones, some might be finished a lot faster.

6.1.1.3. Extended OOPSI-Model: Summary

The proposed Extended OOPSI-Model should be used as a workshop technique to guide the process of finding the highest value in a structured way. It can be used in a workshop to determine an initial roadmap with the customer as well as for detailed discussions about specific scenarios. This makes it an extremely powerful approach. However, the principle of Divide & Conquer should not be applied beyond the point where it adds no value to the discussion. Thus, the conversation should only be split into smaller parts using a child OOPSI-model if the complexity seems to be high enough. It is not possible to give a detailed rule when this point is reached, as it is hard to compare projects and teams with each other. Teams should be encouraged to try and find this point for themselves. The basic idea of the Extended OOPSI-Model, splitting the discussions of a workshop into smaller parts, is summarised in Figure 19.

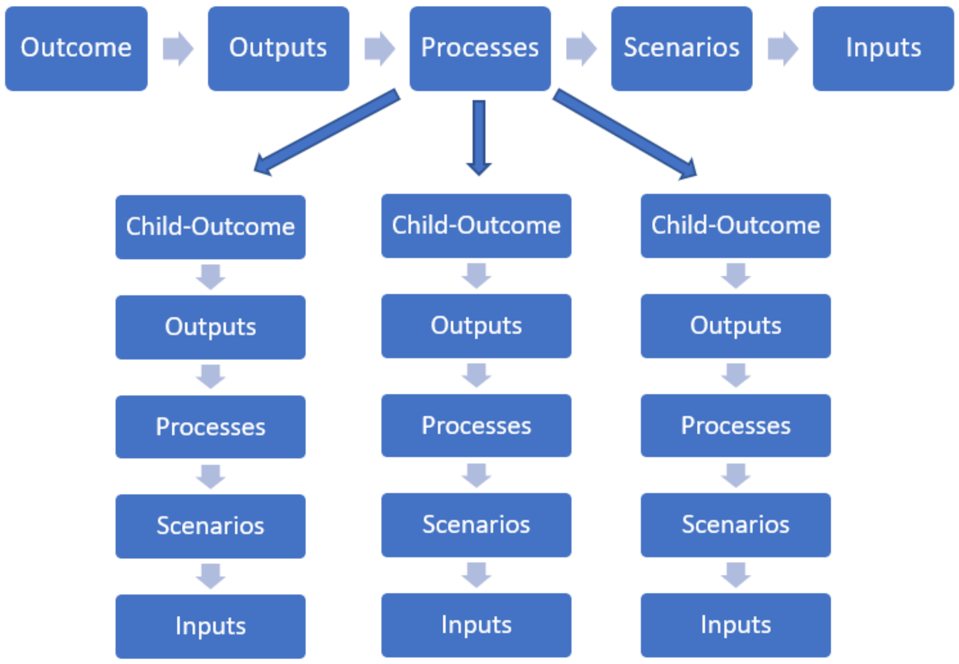


FIGURE 19: SUMMARY EXTENDED OOPSI-MODEL

The main benefit this approach brings, compared to the original OOPSI-Model, is that there is a process on how the discussions can be divided into smaller parts and hence be discussed independently. This helps to formulate the overall picture as well as the most important goals in an understandable way. Due to the connection on the process chain, the link to the main goal is always provided. Hence, the subsequent can be identified and used as pre- and postconditions in the resulting scenarios. Moreover, it encourages the integration of other ideas, such as Story Mapping and Data Personas.

6.2. Automation of Requirements

As the second part of the concept, the following section presents a framework with which the requirements can be continuously checked. The basis for this framework is the Cucumber framework, as introduced in chapter 5.1.1. It interprets the requirements in Gherkin. For simulating a user's behaviour in a browser Puppeteer is chosen.

6.2.1. Cucumber-Puppeteer framework

As the basis for the framework, Cucumber is chosen as the tool of choice because it puts a lot of emphasis on the formulation of requirements. This forces the team to focus more on the requirements themselves. Gauge on the other hand gives the development team more freedom at the definition of the requirements. Feature-wise, both frameworks offer comparable opportunities. Thus, from a technical point of view either one would be a possible choice. However, the fact that the formulation of requirements in Gherkin gives more structure to the requirements definition is the major reason Cucumber is selected [62,64,77].

For interacting with the browser, Puppeteer has been chosen mainly because of one reason. The Specialist for E-Commerce-Platforms hardly utilised the biggest advantage all Selenium-based frameworks bring with them while using the WebdriverIO framework: cross-browser testing. Only 1 out of 26 projects that used automated browser tests executed them on more than one browser. This means that this feature was neglected almost completely. If Puppeteer is now compared with any of the Selenium-based frameworks it can offer higher reliability.

The selection of Puppeteer implies that the resulting framework has to be written in JavaScript. Therefore, the JavaScript implementation of Cucumber, Cucumber-JS, is used. As a foundation for the framework *“Node.js”* and *“npm”* are used. *“Node.js”* is a JavaScript runtime that builds the basis for executing the test framework. Additionally, *“npm”* provides further functionalities that are needed for the test framework. It includes a package manager for *“Node.js”*, a registry with a collection of all needed dependencies as well as a command line interface. This foundation provides the required infrastructure as well as all needed frameworks, including Cucumber-JS and Puppeteer themselves [90,91].

The framework itself also tries to follow the principle of simplicity, similar to Puppeteer. Thus, only a minimal set of dependencies is injected. This means it is designed in a way that it is possible to update each part independently. Hence, it is possible to update the framework with minimal effort.

6.2.1.1. Cucumber-Puppeteer framework: Detailed Description

As already mentioned above, the framework is designed based on the principle of simplicity. As npm provides the functionality of running scripts, it is used to execute Cucumber-JS as test runner. In such a script all possible command line interface options provided by Cucumber-JS can be used, as already explained in section 5.1.1. One of the most important of these options for this framework is the *"world-parameters"* option. It is used to determine an environment used for the execution of the test run. A possible example could be to specify the server on which the tests are executed. In Listing 14, an example is depicted how the tests could be executed on a local environment.

```
"test-all-local": "node_modules/.bin/cucumber-js
     --world-parameters {\\\"env\\\":\\\"local\\\"}"
```

LISTING 14: EXAMPLE NPM SCRIPT TO RUN CUCUMBER-PUPPETEER FRAMEWORK

For the creation of the environment itself a hierarchical structure is provided, that mitigates duplication of environment specifications. As a foundation a default configuration is introduced that holds all basic settings needed by the test framework. This default configuration is shown in Listing 15. It holds the default configuration for Puppeteer as well as a section for settings and fixtures that could be used throughout a test execution.

```
exports.config = {
    puppeteerConfig: {
        headless: false,
        ignoreHTTPSErrors: true,
        defaultViewport: null,
        args: ['--window-size=1600,1000']
    },
    settings: {
        baseUrl: '',
        googleBaseUrl: 'https://www.google.com'
    },
    fixtures: {
        testfixture: "default"
    }
};
```

LISTING 15: DEFAULT CONFIGURATION FOR CUCUMBER-PUPPETEER FRAMEWORK

This default configuration can be extended and overwritten by any of the defined environments in the world-parameters passed by the npm script. For each of these environments a configuration file has to be created similar to the example shown in Listing 15. It is also possible to define multiple environments, whereas always the rightmost environment does have the highest priority. A possible example would be defining the environment as *"local-headless"*. This would end in a merge order where at first the environment *"local"* is merged in the default configuration. The resulting configuration is then again overwritten by the environment *"headless"*. This mechanism provides various possibilities to define the test execution in a flexible way to execute the same set of tests on various systems.

Additionally, it is possible to define a default profile containing multiple command line options, that are used to execute the tests. This is done by specifying those options in a *"cucumber.js"* file. The configuration is used to specify that a report should be created after each test execution. Additionally, a specific format for the produced output in the command line interface is defined. This default profile is shown in Listing 16.

```
const common = [
    '-f node_modules/cucumber-pretty',
    '-f json:reports/cucumber.json'
].join(' ');

module.exports = {
    default: common
};
```

LISTING 16: CUCUMBER-JS DEFAULT PROFILE

To indicate that a Cucumber feature or scenario must interact with a browser, a tagged hook is used. This is used to setup and teardown the browser object, as already suggested in section 5.1.1. If now any feature or scenario uses the tag *"@UI"*, the framework creates a Puppeteer browser object based on the defined environment configuration. This browser object, as well as an initial page are then stored in the world object of the scenario. Hence, all step definitions can access them. Furthermore, all needed pages for the test are created and stored in the scenario's world. This happens before each scenario is executed to ensure there are no dependencies or conflicts between the different scenarios. The implementation of this tagged hook is shown in Listing 17.

```
Before({tags: "@UI", timeout: 15 * 1000}, async function () {
    //launch the browser based on the defined config
    this.browser = await puppeteer.launch(this.environment.puppeteerConfig);
    this.page = await this.browser.newPage();
    await this.page.setCacheEnabled(false);

    // register all available pages for the scenarios,
    // so all are available in the cucumber-world
    this.homePage = new HomePage(this.page);
    this.loginPage = new LoginPage(this.page);
});
```

LISTING 17: TAGGED HOOK FOR PUPPETEER SETUP

As already mentioned in chapter 5, using so-called page objects is a viable solution to organise a test suite to keep the maintenance effort as small as possible. This concept is also considered for this framework. Although it is possible to define all browser operations directly in the step definitions, it is advised to do that in the page objects. This additional layer of abstractions allows to decouple the browser actions from the test logic itself, which makes it easier to maintain. Another important detail of this model is, that the page object itself requires the Puppeteer page object in order to interact with the browser [92].

To make the structure of a web site even more modular, the possibility of using components inside page object is considered. Both objects are derived from the same base class. This *“Base”* class holds several functionalities that can be used by any of the derived objects. The actual pages can then hold any component. An illustration of this architecture is shown in Figure 20.

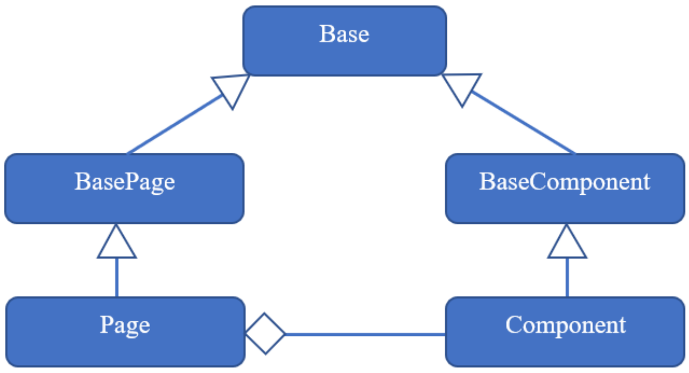


FIGURE 20: ARCHITECTURE OF PAGES AND COMPONENTS

A typical example for this architecture is the integration of a header component in any of the pages containing it. In Listing 18 an exemplified implementation of a page object is shown. The page in the example is derived from the BasePage object and contains a header component.

```
const BasePage = require('../Base.page');
const Header = require('../component/Header.comp');

module.exports = class LoginPage extends BasePage {
    setPage(page) {
        super.setPage(page);
        this.header = new Header(page);
    }

    get loginForm() {
        return '#loginForm';
    }

    get loginFormSubmitButton() {
        return this.loginForm + ' button[type=submit]';
    }

    //------------------------------------------------------------------------
    async isLoginFormShown() {
        return await this.isElementVisible(this.loginForm);
    }

    async fillLoginForm(data) {
        await this.fillForm(this.loginForm, data);
    }

    async submitLoginForm() {
        await this.page.click(this.loginFormSubmitButton);
    }
};
```

LISTING 18: EXAMPLE OF PAGE OBJECT USING A COMPONENT

In the step definitions all functionalities from the page object as well as all from the contained header object can be used. This is illustrated in Listing 19.

```
When('The user enters login credentials', async function () {
    await this.loginPage.fillLoginForm(this.fixtures.forms.login);
    await this.loginPage.submitLoginForm();
});

Then('The user should be logged in', async function () {
    expect(await this.loginPage.header.isLoggedInTextShown()).to.equal(true);
});
```

LISTING 19: EXAMPLE OF STEP DEFINITION USING PAGE OBJECT

After all steps of a scenario are executed, or any of them failed, another hooked tag is triggered. This hook checks the result of the scenario before the browser object is closed. If a scenario fails, a screenshot of the current browser page is added to the Cucumber report. This implementation is illustrated in Listing 20. The screenshot in the report makes the investigation of failed test runs easier.

```
After("@UI", async function (scenarioResult) {
    if (scenarioResult.result.status === 'failed') {
        console.log('Scenario failed: make a screenshot');
        let screenshotFile =
            constants.paths.screenshotDir + '/screenshot_' + Date.now() + '.png';
        await this.page.screenshot({path: screenshotFile });

        let image = fs.readFileSync(screenshotFile);
        // convert binary data to base64 encoded string
        let imageString = new Buffer.from(image, 'png').toString('base64');

        await this.attach(imageString, 'image/png');
    }

    await this.browser.close();
});
```

LISTING 20: TAGGED HOOK FOR PUPPETEER TEARDOWN

In all the examples shown above the keywords *“async”* and *“await”* are used any time a browser interaction is made. This is needed because all Puppeteer functionalities are executed asynchronously. The pair async/await is one way to handle asynchronous JavaScript code by defining and waiting for JavaScript promises. *“Async”* before the function definition makes the function always return a promise indicating the function is executed asynchronously. Additionally, it enables the usage of the keyword *“await”*, which indicates that the code execution will be stopped until a promise is returned by the called function. This means, that each Puppeteer function must be awaited to continue with the test execution. However, this is handled automatically by the async/await functionality. Hence, Puppeteer can test a web application without having to use any sleep-functions that would force the test execution to be idle, even if the test execution is able to continue. Furthermore, it clearly indicates the usage of asynchronous functionalities which makes it easier to read. Cucumber-JS supports asynchronous functionalities as well, otherwise the usage Puppeteer would not be possible. This architecture makes the framework easy to read and fast in comparison to letting the test run rest for a fixed amount of time [77,93].

6.2.1.2. Cucumber-Puppeteer framework: Verification

The purpose of the proposed test framework is to verify that the specified requirements are met, and the result is documented for every test run. However, this test framework is also a software project and thereby is subject to continuous change itself. Thus, all the functionalities of the framework must be tested as well.

For this purpose, *"Jest"*, a framework for Unit Testing in JavaScript, is introduced. It is used to test small parts of the test framework, such as verifying that the building of the environments is working as expected. Such a testcase is shown in Listing 21 [94].

```
test('no environment specified - default should be set', function() {
    let environment = undefined;
    let result = buildEnvironment(environment);

    expect(result.envNames.length).to.be.equal(1);
    expect(result.envNames.includes('default')).to.be.equal(true);

    expect(result.fixtures).to.be.not.equal(undefined);
    expect(result.puppeteerConfig).to.be.not.equal(undefined);
    expect(result.settings).to.be.not.equal(undefined);

    expect(result.fixtures.testfixture).to.be.equal('default');
    expect(result.settings.baseUrl).to.be.equal('');
    expect(result.puppeteerConfig.headless).to.be.equal(false);
});
```

LISTING 21: EXAMPLE OF USING JEST FOR A JAVASCRIPT UNIT TEST

Furthermore, Cucumber-JS is used to test the framework on a higher level. Thus, a small set of Functional Tests is defined to ensure all basic functionalities of the framework work as expected. The created Functional Tests include the creation of a test report in all possible cases as well as the presence of a screenshot if a UI Test failed. The definition of the mentioned scenarios is shown in Listing 22.

```gherkin
Feature: Framework Execution

  This feature describes the basic functionalities of the test framework.

  Scenario: Execute Standard Testcase
    When a test case is executed
    Then a cucumber-report should be created
    And the cucumber-report should contain the scenario with the name 'Today is or
        is not Friday'

  Scenario: Execute Successful UI Test
    When a successful UI test is executed
    Then a cucumber-report should be created
    And the cucumber-report should contain the scenario with the name 'UI-Test
        succeeds'

  Scenario: Execute Failing UI Test
    When a failing UI test is executed
    Then a cucumber-report should be created
    And the cucumber-report should contain the scenario with the name 'Screenshot
        for failing Scenario'
    And the cucumber-report should contain a screenshot
```

LISTING 22: CUCUMBER FEATURE FOR FUNCTIONAL TESTING OF THE PROPOSED FRAMEWORK

6.2.1.3. Cucumber-Puppeteer framework: Integration

As explained in section 2.5, the scenarios should drive the development process. Therefore, they should be integrated in the code base of the Cucumber-Puppeteer framework first. This enables the development team to work on them. Initially, the first scenarios fail directly after their integration. Thus, the desired functionality must be implemented and refactored until all scenarios pass. After some of the functionalities have been added according to that process, a set of Functional Tests is acquired automatically. By using Cucumber-JS as the foundation for the proposed framework, the team documents the requirements in a way that is easy to understand for each member of the development team. Thus, it is also easier to react to changing requirements, as the behaviour is documented unambiguously. Therefore, only the existing scenarios must be changed or extended.

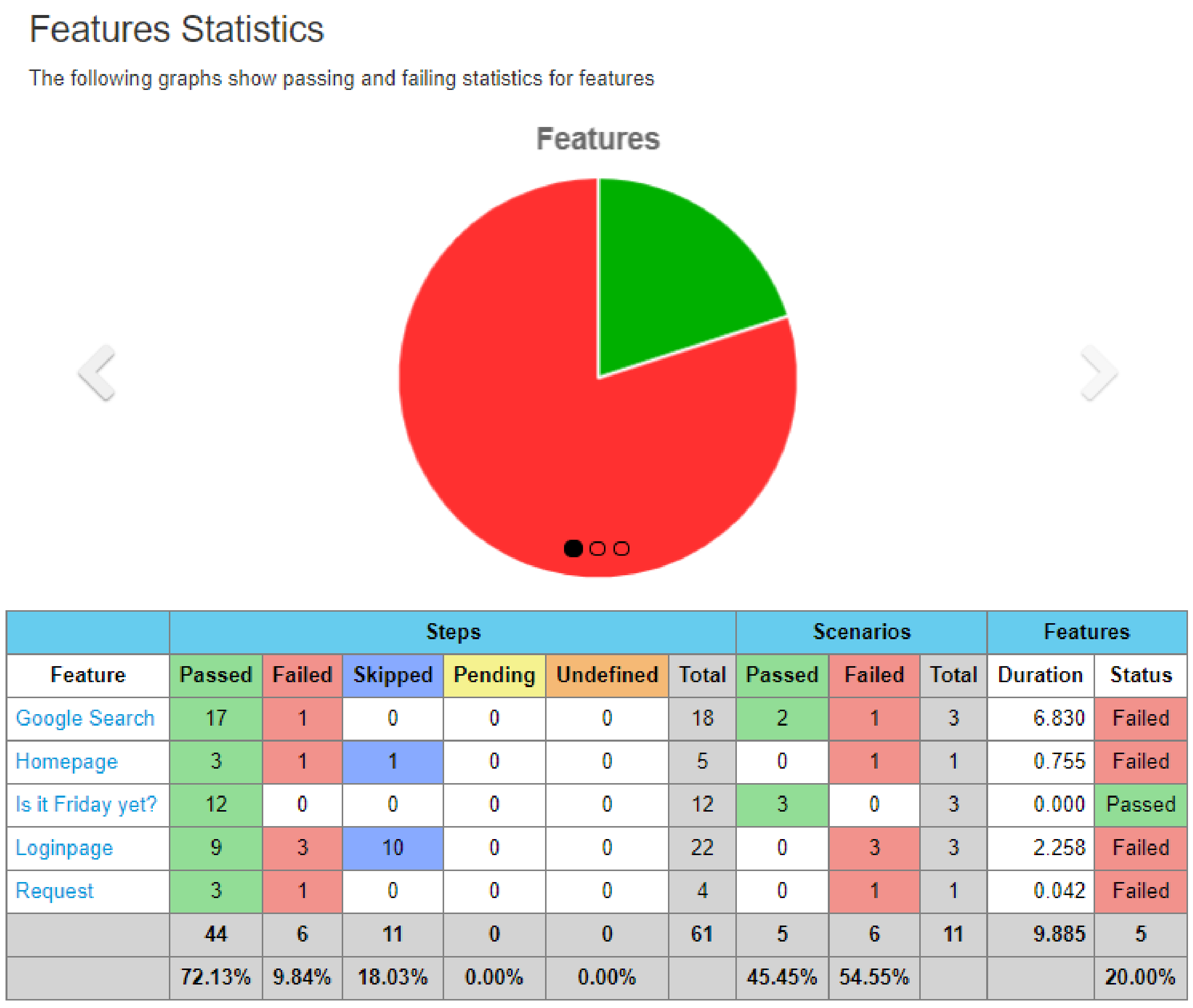
For any following addition to the projects code base, the members of the development team must assure that no existing functionality is broken. Before any new line of code is added, the developers must run the complete set of tests of the Cucumber-Puppeteer framework on their local environment. If any of the scenarios is failing, the developers are not allowed to request the introduction of their changes in the project’s code base. Additionally, the Cucumber-Puppeteer framework should be integrated in the Continuous Integration pipeline of the

Specialist for E-Commerce-Platforms. It should be executed each time a new version of the system is deployed to the test environment. This should be done automatically at least once a day and should ensure that after each change a version is available, that meets the defined requirements.

Inside the organisation of the Specialist for E-Commerce-Platforms Jenkins is used to build the Continuous Integration pipeline. Therefore, this tool is the target for the integration. Jenkins is an automation server, that is capable of building, executing and testing any kind of application. Thus, the proposed integration must be supported by Jenkins [95].

For the integration in this pipeline, Docker is used. The Docker environment provides an environment to run applications inside a container to make it execute independently from the used operating system and infrastructure. Thus, the Cucumber-Puppeteer framework can be packed and executed inside such a container. This architecture allows a software development team to port the framework on any system regardless of the underlying infrastructure. Nevertheless, the framework can still be executed directly on the developer's environment without using a Docker container [96].

In addition to being able to run the framework in a Docker container, it must be ensured that all test runs are properly documented. As already explained earlier, a report is created after each test execution. This report is visualised by a Jenkins plugin, the *"Cucumber Reports Plugin"*. It produces an overview of the executed specifications as well as a detailed description of each scenario. The reports itself allows the development team, as well as any other stakeholder, to check which scenarios failed. A screenshot of the overview of such a report can be seen in Figure 21 [97].



Features Statistics

The following graphs show passing and failing statistics for features

	Steps						Scenarios			Features	
Feature	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
Google Search	17	1	0	0	0	18	2	1	3	6.830	Failed
Homepage	3	1	1	0	0	5	0	1	1	0.755	Failed
Is it Friday yet?	12	0	0	0	0	12	3	0	3	0.000	Passed
Loginpage	9	3	10	0	0	22	0	3	3	2.258	Failed
Request	3	1	0	0	0	4	0	1	1	0.042	Failed
	44	6	11	0	0	61	5	6	11	9.885	5
	72.13%	9.84%	18.03%	0.00%	0.00%		45.45%	54.55%			20.00%

FIGURE 21: EXAMPLE OF A CUCUMBER-REPORT IN JENKINS [97]

6.3. Integration of the Proposed Concept

After describing the two parts of the proposed concept individually, both should be integrated in the development process of the Specialist for E-Commerce-Platforms. As soon as both parts of the concept are implemented, the essential elements of Behaviour Driven Development are in place. As the Specialist for E-Commerce-Platforms uses Scrum as the preferred software development process model, the implementation proposal relies on that. Based on the use cases mentioned by Gojko Adzic in section 2.5 the integration of the concept is considered for the following cases:

1. Integration in a new project
2. Integration in an existing project

6.3.1. Integration in a new project

At the beginning of a new project, the first measure to implement the proposed concept is to use the Extended OOPSI-Model as the preferred workshop technique. An initial set of requirements should be discovered in a first workshop. During this workshop a roadmap should be derived from the initial goal. The most important process steps and scenarios should be pictured in it. Based on this initial roadmap, the involved stakeholders should describe the key scenarios first and provide enough inputs to enable the development team to start working. Furthermore, all stakeholders should plan additional workshops to ensure having enough scenarios ready for development for the following iterations. As the Specialist for E-Commerce-Platforms usually works with Scrum, it is advisable to always have at least enough scenarios specified to fill the next two Scrum Sprints. To have reasonable working packages that can be estimated and planned by a Scrum team, it is recommended to group the scenarios together under a specific User Story, just like the Extended OOPSI-Model suggests. If the team struggles to finish the User Story in a Sprint it should be considered to split the User Stories in multiple smaller ones in future. Thus, the development team should be able to deliver finished working packages each Sprint. The size of these packages must be determined by the team over the duration of some sprints.

Additionally, the framework must be added to the project's code base and Continuous Integration pipeline to test the application continuously. Before the team starts with the actual development of the introduced scenarios, they should be added in feature files. The team should then proceed with the BDD-Cycle as introduced in chapter 2.5.

6.3.2. Integration in an existing project

For software development teams, that are working on an existing project, the integration of Behaviour Driven Development should be tried differently. At first, it should be considered to add the Cucumber-Puppeteer framework as the used tool for Functional Testing. If there is already a framework used for that purpose, the existing tests should be migrated. By doing so, the software development team should get used to the new way of documenting requirements. If no tool is used for Functional Testing, the development team should integrate the Cucumber-Puppeteer framework to describe the most important use cases of the legacy requirements. Over time the team should realise, that the requirements must be formulated in Gherkin to be automated by the Cucumber-Puppeteer framework. Thus, the integration of the Extended OOPSI-Model could provide requirements in the right format for new requirements.

Once this transition has been done, the process should not differ from that of a team that started with the proposed concept right at the beginning of the project, as introduced in chapter 6.3.1. This means that new requirements should be formulated as User Stories that are described by a set of scenarios with real inputs. These working packages should then be estimated and planned according to the development process of the team.

7. Discussion

The following section describes the efforts to introduce the proposed concept inside the organisation of the Specialist for E-Commerce-Platforms as well as the identified limitations.

7.1. Requirements Engineering

As stated in chapter 6.3, a possible first step of the integration of the concept is to introduce the Extended OOPSI-Model as the preferred workshop technique. This should ensure, that all involved stakeholder have the same understanding of the specified requirements. Thus, a training workshop was performed to describe the concept of this workshop technique.

The participants of the workshop had different roles inside the organisational structure of the Specialist for E-Commerce-Platforms. Thus, a mixture of business stakeholder, developers and other roles has been present at the training. After an initial description, the participants were split into teams of four people to ensure that each team had at least a representative of the Three Amigos, as suggested in chapter 4.4. Afterwards the teams had to apply the introduced concept directly on an example tailored to their field of expertise.

All the participants stated that the model helped them to find a common understanding of the stated goal. Thus, each team was able to formulate scenarios, enriched with detailed inputs, to describe the most important business cases of the given problem. One point highlighted by the participants of the workshop was that the discussions were of high quality, because the people in the teams came from different fields. This undermines the importance of the presence of the Three Amigos during such a workshop, as already stated by Martin and Buckney [58].

The hardest part for the workshop participant was to define the outputs within the second step of the proposed concept. Almost all participants mentioned, that it was hard to define any outputs before thinking of the actions producing them. Nevertheless, once the processes of the model had been defined, the teams were able to assign the outputs to the corresponding processes and could proceed with the discussion. The reason for that could be, that this reverse-engineering approach was unfamiliar to them. This should be considered at the introduction of the technique in upcoming training workshops.

Despite the efforts of introducing the Extended OOPSI-Model as a workshop technique inside the organisation, it was not possible to introduce it comprehensively. The introduction of this technique to different teams is still an ongoing approach which might take years to be fully completed. Nevertheless, all meeting participants agreed that the introduction of the Extended OOPSI-Model could solve several existing problems of the currently lived requirements engineering process. They argued, that it might be most useful for specifying complex requirements.

One participant of the training workshop added, that using the proposed model could produce unnecessary overhead. This would apply especially for simple User Stories. Hence, despite the good reasoning of applying the model to complex requirements the usefulness of its general application was questioned. However, contrary to this statement, it is not possible to know the complexity of a User Story in advance, as shown in section 2.2 in an example by Gojko Adzic [19]. Thus, even for seemingly simple requirements, it could be possible that there is not a common understanding between all stakeholders. Using the Extended OOPSI-Model comprehensively should help to mitigate misunderstandings right at the beginning. Such misunderstandings could be very costly. Especially if they are discovered at the end of a project, as already stated by Kent Beck [39]. Therefore, using the Extended OOPSI-Model could be a possible solution to save money over time by preventing misunderstandings at an early stage.

Due to the fact, that the model is not fully integrated in the process of the Specialist for E-Commerce-Platforms it is not possible to show its effectiveness at the time of writing. However, if a development team does have troubles applying the suggested approach, a possible alternative would be a mixture of User Story Mapping and Example Mapping. User Story Mapping tries to keep the big picture in mind while splitting the work into smaller parts. Thus, it might be well suited to get an overview, or a roadmap, of all upcoming goals. Example Mapping could be used to extract detailed scenarios from the resulting User Stories of the User Story Mapping workshop. However, it is argued that the Extended OOPSI-Model brings the flexibility and structure that is needed to guide the requirements engineering process properly. Investigating this comparison is beyond the scope of this thesis, but future studies could take this as a starting point.

7.2. Automation of Requirements

As already stated in section 7.1, neither the Extended OOPSI-Model, nor any alternative, is used comprehensively inside the organisation. Hence, the basis for the introduction of the Cucumber-Puppeteer framework has not been built yet, as no examples are formulated in the requirements engineering process. Thus, it was not possible to gather data that showed how the Cucumber-Puppeteer framework was used to support the development process of previously specified scenarios.

Nevertheless, as already stated in section 6.3, the introduction of such a tool could be a possible first step to introduce Behaviour Driven Development in a team. Due to the lack of available data, it is not possible to confirm this hypothesis in the selected environment at the time of writing. However, the tool has been introduced to one software development team. The integration of the framework inside the whole organisation is an ongoing process that should be evaluated in a following case study. This study should focus on the implication of the framework on the software development process of a software development team. It should consider several teams to confirm or contradict the stated hypothesis for the organisation.

Additionally, to the integration in one development team, the framework was used to describe and test some basic functionalities of the standard solution used by the Specialist for E-Commerce-Platforms as a starting point for new projects. This set of test cases should ease the introduction of the framework to any new project. During the creation of these specifications, some observations have been made.

The first big advantage of using Cucumber as a foundation of the framework seemed to be, that it was possible to reuse any of the created step definitions in different scenarios. Thus, the same test logic could be used, which helped to reduce the amount of code necessary. However, after the creation of the first few scenarios, several of them had to be changed because they were describing the same action. Hence, the first learning was to continuously refactor the used steps, as some of them could have the same meaning. After this first refactoring, even more step definitions could be removed. As this became obvious after only a few scenarios have been created, proper management of the used steps throughout the whole test suite could be one major challenge of using the Cucumber-Puppeteer framework.

Using Puppeteer for simulated the behaviour of a user has been the right choice. It provided all needed functionalities to operate the browser in a fast and stable way. Especially, encapsulating the browser functionalities in several page objects facilitated the management of all browser related actions. This observation is in line with the experiences made by Emily Bache and Geoffrey Bache, as stated in chapter 5 [61]. The additional integration of components, that can be reused on different pages, made it even easier to avoid any code duplications. By using these objects, it was possible to build an abstract representation of the web application under test.

An adaption to the proposed framework can be considered regarding its integration in the Continuous Integration pipeline. It could be an improvement to only introduce code changes as soon as all specifications are executed successfully in a separate build. A possible way to approach that is to deploy the whole application inside an own Docker container after every code change, as introduced in section 6.2.1.3. After the deployment in the Docker container, all specifications should be executed against this version. If this test execution succeeds, the changes can be merged to the code basis automatically. This change would imply that all changes are tested once more in a consistent environment before they are introduced to the code base. Compared to only having a nightly run against a test server as suggested in the proposed concept, this change could provide another measure to increase the quality of the product. However, the comparison of this approach would exaggerate the scope of this thesis and should be considered in future research.

Additionally, to the adaption mentioned above, it might also be possible to exchange any of the parts of the framework. Since the functionalities of Gauge and Cucumber are similar, Gauge could be used as well as the tool of choice. The same applies for Puppeteer and Selenium-based frameworks. If a development team prefers to use Selenium, it might be possible to exchange it as well. It could also be investigated in future research which tools should be used for specific use cases.

7.3. Concept as a whole

Additionally, to the implications of the two distinctive parts of the concept, its overall benefit to the environment must be validated after the concept was internalised in the process of the Specialist for E-Commerce-Platforms. However, this investigation would go beyond the scope of this thesis. Data should be collected from multiple development teams who have changed their development process according to the proposed concept. In this study the focus should be set on two points:

- Could the proposed concept help the software development teams to save time over the whole life cycle of the project?
- Could the proposed concept help the team to evolve the process from Continuous Integration to Continuous Deployment?

Both claims should be compared with the currently practiced process by the Specialist for E-Commerce-Platforms as described in chapter 3. Additionally, to the above-mentioned future research questions, the implication on the satisfaction towards the development process of all involved stakeholders could be investigated.

Another issue that has not been addressed in this thesis because of time constraints is the impact of different contract models on the proposed concept. These implications should be examined in future research as well. Especially if any contractual model limits or hinders the integration of Behaviour Driven Development for the environment under supervision.

8. Conclusion

In this thesis a concept is developed that should help a software development team focussing on E-Commerce-Platforms to integrate Behaviour Driven Development as part of their development process. The concept aims at easing the communication and collaboration of all involved stakeholders of a project.

After introducing the basic concept of different software development process models and Behaviour Driven Development, the focus was set on requirements engineering and automation of these requirements. By focussing on those two disciplines, two essential parts of the development process have been investigated in detail. The overview of the different techniques to improve the requirements engineering process provided a foundation for the selection of a specific methodology for the concept. For the test frameworks the same approach was used. Based on the conducted investigations for each of the mentioned focal points and the needs of the Specialist for E-Commerce-Platforms a selection has been made.

Adapting and extending the OOPSI-Model seemed to be a good choice as the workshop technique for this concept. The resulting Extended OOPSI-Model has been used successfully in a first training workshop. Especially for complex requirements, the application of this model was highlighted. Choosing Cucumber and Puppeteer as a foundation for the test automation framework also seemed to be the right choice, as it provides all needed features. A first set of human-readable requirements could be created for the standard solution of the Specialist of E-Commerce-Platforms. Being a piece of software for itself, the framework has been documented and tested with Cucumber. Thus, Cucumber was used to verify the features of the Cucumber-Puppeteer framework. By using this approach, the framework could be adapted and extended with ease.

Nevertheless, each of the selected techniques or frameworks might be exchanged to fit the needs of a development team. The concept only gives a proposal to integrate Behaviour Driven Development in the environment of the Specialist for E-Commerce-Platform. If the core principals of Behaviour Driven Development are still met, any of the shown workshop techniques or test frameworks could be used successfully.

In conclusion, the proposed concept should help a development team to introduce Behaviour Driven Development. This should lead to implementing the desired requirements more efficiently, as they are formulated in an unambiguous way. Furthermore, by building up a living documentation that continuously verifies the project's requirements, the software development teams can respond to any change with more confidence. This should lead the development teams of the Specialist for E-Commerce-Platform to a progression of their lived development process.

8.1. Future Work

Despite the fact, that the proposed concept is not implemented comprehensively at the Specialist for E-Commerce-Platforms, it builds a solid foundation for any future research on that topic. During the discussion of the proposed concept several new research questions have been introduced.

Based on the proposed concept a case study should be conducted to verify if the main goal of the concept was achieved and why it succeeded or failed. The concept itself could be improved to make it more effective, based on the results of the case study.

The proposed concept provides a foundation for future research, as stated earlier. If the proposed concept fails initially, but is adapted and then integrated successfully, the purpose of the concept is fulfilled. As stated right at the beginning of this thesis, the only constant in software development is change. Hence, the need for continuous improvement applies for the used development process as well.

9. References

[1] D. North, Introducing BDD, 2006. https://dannorth.net/introducing-bdd/ (accessed 26 November 2017).

[2] G. Adzic, Specification by Example: How successful teams deliver the right software, 2nd ed., Manning, Shelter Island, N.Y., 2012.

[3] N.B. Ruparelia, Software development lifecycle models: Hewlett-Packard Enterprise Services, in: ACM SIGSOFT Software Engineering Notes, pp. 8–13.

[4] Winston W. Royce, Managing the Development of large Software Systems, in: Proceedings of IEEE WESCON (Nov. 1970). Reprinted in Proceedings of the 9th International Conference on Software Engineering (1987), pp. 328–338.

[5] Phillip A. Laplante, Colin J. Neill, "The Demise of the Waterfall Model Is Imminent" and Other Urban Myths. Penn State University, Queue February (2004). https://doi.org/10.1145/971564.971573.

[6] Kevin Forsberg, Harold Mooz, The Relationship of System Engineering to the Project Cycle, 1991.

[7] tryqa.com, What is V-model- advantages, disadvantages and when to use it? http://tryqa.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/ (accessed 17 October 2018).

[8] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas, Manifesto for Agile Software Development. http://agilemanifesto.org/iso/en/principles.html (accessed 12 October 2018).

[9] Giulio Barabino, Daniele Grechi, Danilo Tigano, Erika Corona, and Giulio Concas, Agile Methodologies in Web Programming: A Survey, in: Agile Processes in Software Engineering and Extreme Programming, Springer International Publishing, Rome, Italy, 2014, pp. 234–241.

[10] D. Moyo, A.K. Ally, A. Brennan, P. Norman, R.C. Purshouse, M. Strong, Agile Development of an Attitude-Behaviour Driven Simulation of Alcohol Consumption Dynamics, JASSS 18 (2015). https://doi.org/10.18564/jasss.2841.

[11] K. Beck, extreme programming eXplained: Embrace change, Addison-Wesley, Reading MA, 2000.

[12] Ken Schwaber and Jeff Sutherland, The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game, 2017.

[13] Paul Klipp, Getting Started With Kanban, 2011.

[14] C. Ebert, P. Abrahamsson, N. Oza, Lean Software Development, IEEE Softw. 29 (2012) 26–31. https://doi.org/10.1109/MS.2012.116.

[15] Henrik Kniberg, Mattias Skarin, Kanban And Scrum making the most of both, C4Media Inc., 2010.

[16] Olsson, H.H., Alahyari, H., Bosch, J., Climbing the Stairway to Heaven., in: Proceedings of the 38th Euromicro Software Engineering Advanced Applications (SEAA) Conference, Cesme, Turkey, September 5-7 (2012).

[17] Helena Holmström Olsson, Jan Bosch, Towards Agile and Beyond: An Empirical Account Towards Agile and Beyond: An Empirical Account on the Challenges Involved When Advancing Software Development Practices, in: Agile Processes in Software Engineering and Extreme Programming, Springer International Publishing, Rome, Italy, 2014, pp. 327–336.

[18] D.C. Gause, G.M. Weinberg, Exploring requirements: Quality before design, Dorset House Pub, New York, NY, 1989.

[19] G. Adzic, Bridging the Communication Gap, 2009.

[20] Bashar Nuseibeh, Steve Easterbrook, Requirements Engineering: A Roadmap, Future of Sofware Engineering Limerick Ireland 2000 (2000) 35–46.

[21] Daniel Reichart, Peter Forbrig, Anke Dittmar, Task Models as Basis for Requirements Engineering and Software Execution, in: Proceedings of the 3rd annual conference on Task models and diagrams, pp. 51–58.

[22] Eric S. K. Yu, Towards Modelling And Reasoning Support For Early-phase Requirements Engineering - Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on, 1997.

[23] IEEE Computer Society, IEEE Recommended Practice for Software Requirements Specifications, 1998.

[24] Mike Cohn, Advantages of User Stories for Requirements, 2004. https://www.mountaingoatsoftware.com/articles/advantages-of-user-stories-for-requirements (accessed 2 July 2019).

[25] Ron Jeffries, Essential XP: Card, Conversation, and Confirmation, 2001. https://ronjeffries.com/xprog/articles/expcardconversationconfirmation/ (accessed 2 July 2019).

[26] Antony Marcano, It starts with a Story…, 2014. http://antonymarcano.com/blog/2014/05/it-starts-with-a-story/ (accessed 11 March 2019).

[27] Antony Marcano, How the industry broke the Connextra Template: Apples, Oranges and User Stories, 2016. http://antonymarcano.com/blog/2016/08/how-the-industry-broke-the-connextra-template/ (accessed 11 March 2019).

[28] Jeff Patton, Story Mapping: discover the whole story, 2015. https://www.slideshare.net/jeffpatton/user-story-mapping-discovery-the-whole-story (accessed 11 March 2019).

[29] William C. Wake, INVEST in Good Stories: The Series, 2017.

[30] Seb Rose, Every Process Needs Thoughtful Participants: P3X - People, Product & Process eXchanges 2018, CodeNode, London, 2018.

[31] Liskin O., Pham R., Kiesling S., Schneider K., Why We Need a Granularity Concept for User Stories, in: Agile Processes in Software Engineering and Extreme Programming, Springer International Publishing, Rome, Italy, 2014, pp. 110–125.

[32] James A. Whittaker, What Is Software Testing? And Why Is It So Hard?: Software testing is arguably the least understood part of the development process. Through a four-phase approach, the author shows why eliminating bugs is tricky and why testing is a constant trade-off. Florida Institute of Technology, IEEE Softw. January/February (2000).

[33] S. Nidhra, Black Box and White Box Testing Techniques - A Literature Review, IJESA 2 (2012) 29–50. https://doi.org/10.5121/ijesa.2012.2204.

[34] K. James Perry Rodrigues, Orville Jay Potter 6067639, 2000.

[35] R. Owen Rogers, Acceptance Testing vs. Unit Testing: A Developer's Perspective, in: Extreme Programming and Agile Methods - XP/Agile Universe 2004: LNCS 3134, Calgary, Canada, 2004, pp. 22–31.

[36] H. Erdogmus, M. Morisio, M. Torchiano, On the effectiveness of the test-first approach to programming, IIEEE Trans. Software Eng. 31 (2005) 226–237. https://doi.org/10.1109/TSE.2005.37.

[37] Thirumalesh Bhat, Nachiappan Nagappan, Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies: Proceedings of the 5th ACM-IEEE International

Symposium on Empirical Software Engineering September 21-22, 2006, Rio de Janeiro, Brazil, Association for Computing Machinery (ACM), New York.

[38] Gáspár Nagy and Seb Rose, The BDD Books - Discovery: Explore behaviour using examples, 2018.

[39] Kent Beck, Test-Driven Development By Example. Three Rivers Institute, 2002.

[40] E. Bjarnason, M. Unterkalmsteiner, M. Borg, E. Engström, A multi-case study of agile requirements engineering and the use of test cases as requirements, in: Information and Software Technology, pp. 61–79.

[41] Robert C. Martin, Grigori Melnik, Tests and Requirements, Requirements and Tests: A Möbius Strip, IEEE Softw. January/February 2008 54–59.

[42] Fitnesse: The fully integrated standalone wiki and acceptance testing framework. http://fitnesse.org/FrontPage (accessed 8 March 2019).

[43] Krzysztof Wnuk, Linus Ahlberg, Johannes Persson, On the Delicate Balance between RE and Testing: Experiences from a Large Company, in: Proceedings // 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), IEEE, 2014.

[44] W. Trumler, F. Paulisch, How “Specification by Example” and Test-Driven Development Help to Avoid Technial Debt, in: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD), IEEE, 2016, pp. 1–8.

[45] Gáspár Nagy, Continuous Behavior – BDD in Continuous Delivery: P3X - People, Product & Process eXchanges 2018, CodeNode, London, 2018.

[46] Chris Matts, The IT Risk Manager: About. https://theitriskmanager.com/about/ (accessed 22 March 2019).

[47] Cucumber Limited, Cucumber. https://cucumber.io/ (accessed 8 March 2019).

[48] J. Patton, Story Mapping Quick Reference. https://www.jpattonassociates.com/story-mapping-quick-ref/ (accessed 2 July 2019).

[49] Java. https://openjdk.java.net/ (accessed 3 April 2019).

[50] JavaScript. https://www.javascript.com/ (accessed 3 April 2019).

[51] WebdriverIO. https://webdriver.io/ (accessed 8 March 2019).

[52] Gojko Adzic, Chris Matts, Feature Injection: three steps to success, 2011. https://www.infoq.com/articles/feature-injection-success (accessed 15 March 2019).

[53] Kent McDonald, Feature Injection Clarified (Hopefully), 2014. https://www.kbp.media/featureinjectionclarified/ (accessed 17 March 2019).

[54] Gojko Adzic, David Evans, Fifty Quick Ideas to improve your User Stories, 2014.

[55] Jim Bowes, An introduction to user story mapping, 2017. https://manifesto.co.uk/user-story-mapping/ (accessed 14 March 2019).

[56] Matt Wynne, Introducing Example Mapping, 2015. https://cucumber.io/blog/example-mapping-introduction/ (accessed 22 March 2019).

[57] Brian Gilmour, Frazer Shaw, Three Amigos in the World of Agile, 2017. http://www.edgetesting.co.uk/news-events/blog/three-amigos-in-the-world-of-agile (accessed 22 March 2019).

[58] Jenny Martin, BDD Discovery and OOPSI, 2016. https://jennyjmar.com/2016/04/16/bdd-discovery-and-oopsi/ (accessed 15 March 2019).

[59] Jenny Martin, OOPSI-Mapping Workshop: P3X - People, Product & Process eXchanges 2018, CodeNode, London, 2018.

[60] Dominique Winter, Jörg Thomaschewski, Eva-Maria Schön, Persona driven agile development: Build up a vision with personas, sketches and persona driven user stories, in: 7th Iberian Conference on Information Systems and Technologies, 2012.

[61] Emily Bache and Geoffrey Bache, Specification by Example with GUI Tests - How Could That Work?, in: Agile Processes in Software Engineering and Extreme Programming, Springer International Publishing, Rome, Italy, 2014, pp. 320–326.

[62] Gauge: Less Code, Less Maintenance, More Acceptance Testing. Gauge is a free and open source test automation framework that takes the pain out of acceptance testing. https://gauge.org/ (accessed 21 May 2019).

[63] Cucumber Installation. https://cucumber.io/docs/installation/ (accessed 21 May 2019).

[64] Cucumber Github. https://github.com/cucumber (accessed 2 July 2019).

[65] Cucumber Introduction. https://cucumber.io/docs/guides/overview/ (accessed 21 May 2019).

[66] Matt Wynne, Aslak Hellesoy, Steve Tooke, Jacquelyn Carter, The Cucumber Book: Second Edition, 2017.

[67] Cucumber Limited, 10 Minute Tutorial. https://cucumber.io/docs/guides/10-minute-tutorial/ (accessed 22 May 2019).

[68] Cucumber World. https://cucumber.io/docs/cucumber/state/#world-object (accessed 31 May 2019).

[69] Gherkin Reference. https://cucumber.io/docs/gherkin/reference/ (accessed 2 July 2019).

[70] Cucumber Limited, Cucumber Reference, 2019. https://cucumber.io/docs/cucumber/api/ (accessed 23 May 2019).

[71] Gauge Github. https://github.com/getgauge (accessed 2 July 2019).

[72] Zabil Maliackal, Why we built Gauge, 2018. https://blog.getgauge.io/why-we-built-gauge-6e31bb4848cd (accessed 14 June 2019).

[73] Markdown. https://www.markdownguide.org/ (accessed 30 May 2019).

[74] Gauge Writing Specificatio. https://docs.gauge.org/latest/writing-specifications.html (accessed 30 May 2019).

[75] Gauge - Getting Started Guide. https://gauge.org/getting-started-guide/ (accessed 31 May 2019).

[76] Selenium. https://www.seleniumhq.org/ (accessed 8 March 2019).

[77] Puppeteer. https://pptr.dev/ (accessed 2 July 2019).

[78] Selenium Documentation; Introduction. https://www.seleniumhq.org/docs/01_introducing_selenium.jsp (accessed 8 June 2019).

[79] Selenium WebDriver. https://www.seleniumhq.org/docs/03_webdriver.jsp (accessed 8 June 2019).

[80] WebDriver, 2019. https://www.w3.org/TR/webdriver/ (accessed 8 June 2019).

[81] Selenium Grid. https://github.com/SeleniumHQ/selenium/wiki/Grid2 (accessed 08,06.2019).

[82] Appium: Automation for Apps. http://appium.io/ (accessed 8 June 2019).

[83] WebdriverIO Selenium Standalone Service. https://github.com/webdriverio/webdriverio/tree/master/packages/wdio-selenium-standalone-service (accessed 8 June 2019).

[84] WebdriverIO Frameworks. https://webdriver.io/docs/frameworks.html (accessed 8 June 2019).

[85] Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/ (accessed 9 June 2019).

[86] Chromium. https://www.chromium.org/Home (accessed 9 June 2019).

[87] Puppeteer for Firefox. https://www.npmjs.com/package/puppeteer-firefox (accessed 9 June 2019).

[88] Chromium Core Principles. https://www.chromium.org/developers/core-principles (accessed 9 June 2019).

[89] John Ferguson Smart, How not to prepare test data in JBehave and Cucumber, 2017. https://johnfergusonsmart.com/givenstories-anti-pattern-not-prepare-test-data-jbehave-cucumber/ (accessed 21 May 2019).

[90] Node.js Foundation, node.js. https://nodejs.org/en/about/ (accessed 15 June 2019).

[91] npm. https://www.npmjs.com/about (accessed 15 June 2019).

[92] Martin Fowler, PageObject, 2013. https://martinfowler.com/bliki/PageObject.html (accessed 16 June 2019).

[93] Async/Await, 2019. https://javascript.info/async-await (accessed 16 June 2019).

[94] Jest. https://jestjs.io/ (accessed 15 June 2019).

[95] Jenkins. https://jenkins.io/ (accessed 3 April 2019).

[96] What is a Container?: A standardized unit of software. https://www.docker.com/resources/what-container (accessed 21 June 2019).

[97] Cucumber Reports Plugin. https://plugins.jenkins.io/cucumber-reports (accessed 21 June 2019).