MASTERING SCRUM

EXPLORING PRACTICAL CHALLENGES AND DERIVING A NOVEL SOFTWARE SOLUTION

Dissertation

In partial fulfillment of the requirements for the academic degree of
Doctor rerum naturalium (Dr. rer. nat.)

Institute of Computer Science
Faculty of Computer Science, Electrical Engineering and Mathematics
Paderborn University

Adrian Hülsmann

September 2024

Adrian Hülsmann
Mastering Scrum:
Exploring Practical Challenges and Deriving a Novel Software Solution
September 2024

ABSTRACT

More than 75% of current software projects utilize agile development methods, with *Scrum* being employed in 80% of these projects, making it the most popular software development framework. Despite its straightforward structure and simple rules, practical implementation issues often hinder the methodology from achieving its full potential, resulting in prolonged project durations and increased development costs.

This thesis is dedicated to this problem. Focusing on analyzing current Scrum project management solutions and identifying their failure to adequately address the issues with Scrum, it proposes a new software solution that fully maps the agile development framework and better supports Scrum teams in overcoming practical implementation problems.

ZUSAMMENFASSUNG

Mehr als 75 % aller Softwareprojekte werden heutzutage mit agilen Methoden durchgeführt. Dabei wird in 80 % der Fälle *Scrum* eingesetzt, welches das weltweit beliebteste Softwareentwicklungsframework darstellt. Durch den einfachen Aufbau und das klare Regelwerk gilt das Scrum-Framework als leicht verständlich. Allerdings zeigen sich in der praktischen Umsetzung immer wieder Probleme, die dazu führen, dass nicht das gesamte Potential der Methodik genutzt wird, wodurch sich Projektlaufzeiten verlängern und damit die Kosten zur Entwicklung von Software erhöhen können.

Diesem Problem widmet sich diese Arbeit. Sie konzentriert sich dabei auf die Analyse heutiger Projektmanagementanwendungen und stellt fest, dass diese nicht in der Lage sind, die Probleme bei der Umsetzung von Scrum adäquat zu adressieren. Aus diesem Grund wird eine neue Softwarelösung vorgestellt, die das agile Entwicklungsframework zur Gänze abbildet und Scrum-Teams bei der Bewältigung der praktischen Implementierungsprobleme besser unterstützt.

ACKNOWLEDGMENTS

This dissertation represents the culmination of years of work. I am deeply grateful to everyone who has contributed to this journey.

First and foremost, I would like to express my profound gratitude to my supervisor *Prof. Dr. Gerd Szwillus*. It was your initial inspiration, back when I was a master's student, that led me into the fascinating world of Human-Computer Interaction. The eight years working with you as a research assistant have been both rewarding and enriching. Whether it was the seemingly endless number of oral student examinations or discussing the intricacies of usability problems, I enjoyed every moment. Giving me absolute freedom to explore research topics that genuinely interested me was a gift that allowed me to grow as a researcher, while your unwavering belief in me and my work has been a constant source of motivation.

I would also like to extend my heartfelt thanks to *Irene Roger*, who, with her incredible knowledge of every detail of university life, was a cornerstone of our research group. I will always cherish the many chats we shared and the invaluable assistance you provided throughout my time as a research assistant.

My gratitude also goes to *Prof. Dr. Stefan Böttcher* and *Prof. Dr. Karsten Nebe*, who kindly agreed to serve as reviewers of this thesis. I deeply appreciate your time, commitment, and willingness to take on this role.

On a more personal note, I want to express my deepest gratitude to my beloved wife, *Lisi*. Your patience, love, and unwavering support have been the bedrock of my life during these challenging years. Without your understanding and encouragement, this thesis would not have been possible. Lisi, you have been my anchor, and I am unbelievably thankful for everything you have done for me.

To my children, *Pamina* and *Alwin*, I owe a debt of gratitude that words cannot fully capture. Thank you for your incredible understanding and patience as I worked on my thesis. Pamina, your encouragement kept me going during the toughest times, and Alwin, your question about what life will be like after I finish my PhD was both profound and eye-opening. Since you were born, you have known me as someone spending all weekends on his "Doktorarbeit," and I am so excited to now have more time with both of you to enjoy life together.

CONTENTS

1 INTRODUCTION

In the dynamic landscape of software development methodologies, agility has emerged as a fundamental paradigm, with *Scrum* standing at the forefront as a widely adopted framework. The iterative and incremental nature of Scrum facilitates adaptability and responsiveness to changes, which frequently occur in nearly any software development project. As a result, Scrum is known for reducing time-to-market and increasing customer satisfaction through an overall higher software quality. However, implementing Scrum often encounters challenges that can disturb the collaborative work environment, causing project prolongations and higher costs, which demand better ways of managing the software development process.

This doctoral thesis delves into the multifaceted realm of Scrum, investigates its challenges, analyzes how popular project management tools relate to the identified issues, and finally presents a novel project management approach utilizing a combination of single and multi-user interfaces meticulously tailored for the intricacies and unique demands of the Scrum methodology.

1.1 MOTIVATION AND PROBLEM STATEMENT

The impetus for this research originally stems from personal observations made during a software development project in 2009 while being enrolled as a Master's student in the computer science program at the University of Paderborn. The project had a dynamic and explorative nature with undefined outcomes and probable directional changes. Hence, it necessitated an agile approach, leading to the adoption of the Scrum methodology, which the supervisors initially perceived as an ideal fit for the project group due to its structured approach and clear rule set.

However, this first personal engagement with Scrum unveiled a series of complexities and a significant disconnect between the theoretical understanding of the framework and its practical implementation. While the rules of Scrum were commonly understood by all team members, the framework was soon modified to what seemed to be a better fit for the circumstances of the group. Initially perceived as beneficial, these modifications gradually led to a myriad of operational issues, including difficulties in managing project requirements, distributing

tasks and responsibilities, and ensuring efficient knowledge transfer within the team. It quickly became evident that the deviations from Scrum's established protocols escalated into complex issues impacting overall project management.

Likewise, this project also unveiled challenges regarding managing the Scrum process via appropriate software. While the group initially agreed to use *Jira*, known as the most widely used software for managing software development projects, it soon revealed its inadequacy in fully supporting integral parts of Scrum. This inadequacy necessitated integrating other more lightweight tools, like spreadsheets and pen-and-paper, to compensate for Jira's limitations. While these supplementary tools addressed certain immediate needs, they introduced new problems, such as the synchronization of important data, so the resultant fragmentation of information significantly slowed down decision-making processes and reduced team efficiency.

These initial observations were later further reinforced by experiences as a research assistant and turned out to be not only isolated incidents but rather widespread issues in both academic and industrial settings. The challenges faced were not unique to the specific context of the project group but reflected a common discrepancy in the application of Scrum, which, among experts, is characterized by being simple to learn but difficult to master.

With more than 75% of software projects utilizing agile development methods and Scrum being employed in more than 80% of these projects [300], the motivation for this thesis arises from a dual recognition of Scrum's pivotal role in modern software development and the diverse challenges experienced during its implementation. These challenges, spanning from team-related dynamics to tool limitations, can limit the methodology from achieving its full potential, causing prolonged project durations and increased development costs. Consequently, they prompt a comprehensive exploration aimed at enhancing Scrum's effectiveness.

1.2 SOLUTION

The overarching goal of this thesis is to contribute to bridging the gap between the idealized Scrum methodology and the real-world challenges encountered during its implementation. In particular, the focus is on better understanding the effects of agile management software on the collaborative development process and deriving improvement potentials for a more effective and efficient Scrum project management practice.

First, this thesis identifies and analyzes the intricate challenges associated with the implementation of Scrum. Based on these findings, it critically examines current Scrum management applications and investigates how their usability, offered functionality, and proposed feature sets relate to the identified problems. Considering the identified limitations of current tool support, this thesis finally proposes a novel project management solution particularly designed for the special demands of agile Scrum teams. The centerpiece of this solution is an "Interactive Scrum Space," a combination of various single and touch-based multi-user interfaces to foster a more engaging and collaborative agile working environment.

The strategic use of these interfaces allows for a more natural and intuitive interaction with project data. By simplifying the process of managing different aspects of a Scrum project, the system enables team members to have a clearer understanding and control over project management tasks. This includes real-time backlog management, dynamic sprint planning tools, and visual representations of project progress, all designed to address the previously identified practical Scrum challenges and to improve team coordination and decision-making.

1.3 THESIS STRUCTURE

Following Chapter 1, representing the current introduction, the structure of this thesis unfolds in three cohesive parts, with each part consisting of three individual chapters.

Part I - Theoretical Foundation

The first part provides the theoretical foundation for this thesis and elaborates on the historical backdrop of computers and electronic computation, the rise of software development, the shift from plan-driven to agile methodologies, and finally, examines Scrum as today's most popular software development framework.

Chapter 2: Historical Background

This chapter begins by tracing the historical evolution of human computers to electronic computation, followed by detailing key milestones, such as the rise of software development, the software crisis, and the establishment of software development as an engineering discipline. Finally, this chapter reflects on the last 50 years of software engineering, extracting valuable learnings for the thesis.

Chapter 3: The Evolution of SDLC Models

Chapter 3 delves into the diverse models of the software development lifecycle (SDLC). It defines the SDLC, clarifies the terminology around

models, methodologies, and frameworks, and explores the evolution from plan-driven development to iterative and incremental approaches and the advent of agile methodologies.

Chapter 4: Scrum - Theory and Practice

This chapter provides an in-depth examination of the Scrum framework, elucidating its roots and underlying theory. It covers the entire ruleset about the Scrum team, roles, backlogs, and events of the sprint cycle and also addresses aspects beyond Scrum, which, although not part of the framework, finally evolved into de-facto standards.

Part II - Problem Analysis

After setting the theoretical foundation, the second part of this thesis focuses on the problem analysis. With that regard, it first postulates the research questions and explains the approach before elaborating on the identified issues and challenges of Scrum. It further investigates how the elements of Scrum are reflected in today's agile project management tools, thus giving an overview of the status quo of Scrum challenges and drawbacks of existing tool support.

Chapter 5: Research Questions and Methods

What challenges do Scrum teams usually face? How are these problems reflected in existing tool support? Moreover, what could a novel tool tailormade for the particular demands of Scrum look like? This chapter elaborates on these research questions, outlines the research methodology, and provides an overview of how the research questions have been approached.

Chapter 6: Scrum Issues and Challenges

This chapter offers a detailed exploration of the challenges within the Scrum framework. Employing diverse research methods, including literature review, ethnographic studies, and interviews, it presents findings on challenges related to team dynamics, knowledge management, product owner roles, and the nuances of Scrum events.

Chapter 7: Status Quo of Scrum Tool Support

Chapter 7 investigates the existing landscape of Scrum tool support. It encompasses a thorough analysis of tool types, usage trends, and limitations of agile Application Lifecycle Management (ALM) tools, shedding light on usability issues and contradictions with the rules of Scrum.

Part III - The Implemented Solution

The third and final part of this thesis focuses on implementing a novel software solution for the previously identified problems of Scrum, thereby putting particular emphasis on designing a system

with dedicated support for the collaborative activities of the Scrum framework.

Chapter 8: Natural User Interfaces in Agile Environments

This chapter explores Natural User Interfaces (NUIs) as a new approach to Human-Computer Interaction (HCI) and introduces the concept of combining multi-user touch-based interfaces with classical single-user desktop interfaces to better support the collaborative activities of the Scrum framework. Delving into different NUI types, design considerations, and related work lays the groundwork for the succeeding chapter.

Chapter 9: Introducing an Interactive Scrum Space

As a culmination of the thesis, this chapter introduces a NUI-driven interactive Scrum space. It outlines the implementation details, features, and solutions addressing the identified challenges. The chapter concludes with an evaluation and critical discussion, paving the way for future considerations.

Chapter 10: Summary and Closing

The final chapter summarizes the key findings, contributions, and implications of this thesis and outlines potential starting points for future research.

Part I

THEORETICAL FOUNDATION

The core of this thesis, a novel software solution for managing the Scrum process, must be set against the backdrop of how software development evolved. By this, the reader will be able to understand core aspects of software development in general and learn how agile methodologies finally displaced traditional processes by incorporating a novel work philosophy.

Central to this discussion is the emergence of Scrum as the leading agile development approach. Scrum will be comprehensively explored, elucidating its values, practices, and distinctive attributes. This detailed analysis sets the groundwork for Part II, focusing on critically examining the issues and challenges associated with Scrum implementation.

2

HISTORICAL BACKGROUND

"Human history in essence is the history of ideas." — H. G. Wells [285]

What is meant by this quote from the British science fiction author *Herbert G. Wells* (1866 - 1946) is that the explanation of history cannot be reduced to the impacts of outstanding individuals and their exceptional characters, acting as heroes with superior intellect and divine inspiration for mankind - also known as the "Great Man Theory," which is associated with the Scottish historian *Thomas Carlyle* [48]. Instead, history is shaped by a complex interplay of different ideas collectively intertwined with varying views about culture, morality, authority, etc., all of which must be considered against their time.

As a result, every topic of discussion incorporates its own history, a course of failure and success that has led to the actual point in time where it becomes reinvestigated to set the stage and determine its future path. To decide on the next steps, it is essential to consider and learn from the past to assimilate success factors and what has been achieved and understand the stumbling blocks and things to avoid for not making the same mistakes again.

For this thesis, it therefore makes sense to first look at the historical background of software development in general before investigating the issues and challenges of Scrum as today's leading software development process. This historical evolution should provide insights into the overall programming culture, culminating in the principles, techniques, and patterns used today, and explore origins that might be less familiar, thus increasing cross-cultural awareness and understanding of the domain.

However, in the words of *Barry Boehm*, who, as will be shown in the further course, is one of the many outstanding persons contributing to the evolution of software development: "One has to be a bit presumptuous to try to characterize both the past and future of software engineering in a few pages" [32]. Hence, this historical review does not claim completeness but is instead intended to illustrate the milestone ideas and collective transformative power for evolution in the field starting with times in which "computers" were not even associated with machines, to today's modern "agile" development processes.

2.1 FROM HUMAN COMPUTERS TO ELECTRONIC COMPUTATION

"With the rapid development and widespread use of electronic computers, there is a tendency to forget that all computations were formerly done by hand; computing a verb, has become computer, a noun."

— Beverly E. Golemba [97]

2.1.1 *17th Century - WW II: Human Computers*

The word "computer" is particularly interesting since its meaning has changed dramatically in the last few hundred years. From its origins in Latin in the middle of the 17th century, it meant "someone who computes" and remained associated with human activity for over 300 years before it became applied to the first electronic computation devices [220, p. 8]. Alternatively, put in the words of the expert in computing and society, *David Alan Grier*: "Before computers were machines, they were people" [102].

2.1.1.1 *Comet Halley and the First Division of Mathematical Labor*

The story of human computers begins with the computation of the orbit of *Comet Halley*, soon after the invention of calculus in the mid-17th century by *Issac Newton* (1642 - 1727) and *Gottfried Leibniz* (1646 - 1716).

In 1705, the astronomer and mathematician *Edmund Halley* (1656 - 1742) published "A Synopsis of the Astronomy of Comets" [109], in which he used Newton's laws of motion and the new calculus to compute the periodicity of the comet that later should be named after him. While he realized that this comet's orbit was influenced by the mutual interactions of the sun and the planets Saturn and Jupiter, he worked hard to find a simple mathematical expression for this interaction. However, he ultimately failed, finally having only a crude approximation of the comet's orbit [102].

Some years later, the French mathematician *Alexis-Claude Clairaut* (1713 - 1765) created a new mathematical model for the orbit of Comet Halley. However, it could only be solved numerically [102]. As a result, in 1758, he constituted a team to undertake the calculations and recruited two mathematically skilled friends: the astronomer *Joseph Jerome Lalande* (1732 – 1807) and *Nicole-Reine Lepaute* (1723 – 1788), the wife of a royal clockmaker in the Luxembourg Palace.

For almost half a year, the three friends met at "a common table in the Palais Luxembourg using goose-quill pens and heavy linen paper" [103, p. 20] and obsessively computed the comet's orbit under

the gravitational pulls of Jupiter and Saturn by breaking down the complicated math into an extraordinary series of mini-steps [254]. After completing their work, they announced that the comet would reach its perihelion in the following year, on April 13th, 1759 [102]. As it turned out, they missed the true perihelion by 31 days[1]. Therefore, they earned much criticism by one of the great intellectuals of that time, *Jean-Baptiste le Rond d'Alembert* (1717 – 1783), who ridiculed Clairaut's work as being "more laborious than deep" [254] and argued that computation was not a proper substitute for careful analysis [102].

Using division of labor for calculating the orbit of Comet Halley

Apart from their calculation being a tenfold improvement over Edmund Halley's prediction, the more important innovation was the "division of mathematical labor, the recognition that a long computation could be split into pieces that could be done in parallel by different individuals" [103].

That is why, against the doubts of d'Alembert, others began to organize computing groups very soon after the return of Comet Halley, and in that same year, Lalande and Lepaute were commissioned by the French *Academie of Sciences* to compute the nautical almanac "Connaissance des Temps" [254].

2.1.1.2 *The Division of Labor and Shift in the Meaning of Computation*

The concept of division of labor gained more momentum after the English economist *Adam Smith* (1723 - 1790) published his magnum opus called "The Wealth of Nations" in 1776 and identified division of labor and specialization as central elements of productive power [102].

"Human computers" and the division of mathematical labor gained momentum

Based on Smith's ideas, the French mathematician *Gaspard de Prony* (1755 - 1839) started a monumental project of calculation on behalf of the French government to create seventeen volumes of trigonometric and logarithmic tables, supposed to establish the metric system and unify the multiple measurements and standards used throughout the nation after the French Revolution.

Using a hierarchical "divide and conquer" approach of human computers, he divided the complex computational tasks into a series of additions and subtractions [102]. On top of the hierarchy, a handful of "excellent mathematicians" conceived analytic formulae and strategies for the calculation. Below them, he installed eight "calculators" with knowledge in analysis "who would deduce from these formulas the numbers needed to begin actual computations" [69], while at the lowest level, up to eighty persons with only basic knowledge of arithmetic finally performed millions of the elemental additions and subtractions.

1 As known today, the calculations did not consider the influences of Uranus and Neptune, which had not been discovered in 1757. [103]

However, even with the division of labor and "manufacturing methods," as Prony later called them [69], it required nearly six years to complete their work, which he described as not only leaving "nothing to desire with respect to exactitude, but the most vast and imposing monument to calculation ever executed or even conceived" [69].

Prony's tables and labor division marked a shift in social perception of human computers. While in the eighteenth century, computation was put on a level of "intelligence" and as the distinctive activity of philosophers, scientists, or mathematicians, it drifted by the turn of the nineteenth century to the very opposite, namely into that of dull, repetitive, and poorly paid bodily labor. Prony's calculation project brought together classes of people that seemed to be immiscible. From mathematicians with profound knowledge of analysis to numerous anonymous people working as human computers, knowing and performing only the crudest rules of arithmetic. In the end, computation lost its glory and was pushed away from "intelligence" towards "work" [69].

Women took the positions of human computers

Against this backdrop, it comes as no surprise that from then on, primarily women, far from being socially treated as having equal mental performance as men, took the positions of human computers and performed the factory-style of computing for astronomical, statistical, and military projects until the end of World War II.

2.1.1.3 *Human Computers in War Times and the Particular Role of Women*

By the early twentieth century, computing was considered women's work[2]. The First and Second World Wars further changed the job demands of computation [159]. With men in war, the military recruited large numbers of female human computers, many with a college education, who should calculate (by hand) a variety of military problems, ranging from navigation tables, artillery trajectories, ballistics for anti-aircraft munitions[3] to shock wave propagation, stress on airframes, efficient bombing plans, radar reflections, optimal production strategies, and likely cipher keys [102].

One example is the *National Advisory Committee for Aeronautics* (NACA)[4] of the United States, which was in urgent need during World War II to develop new aircraft that were faster and safer than the existing ones. Many mathematically trained women were hired at the *Langley Memorial Aeronautical Laboratory* as human computers for

2 This is also shown by the fact that in later years, the problem-solving horsepower of early computing machines was approximated in "girl-years" or units of "kilo-girl." [254]

3 A famous example of such work is the "Aberdeen Proving Ground" in Maryland, USA, which is described as "the Manhattan Project of its day." [254]

4 NACA was founded in 1915 and renamed in 1958 into the nowadays well-known NASA (National Air and Space Administration).

calculating aeronautical research data, such as the first wind tunnel experiments [97].

Another famous example of the application of human computers during World War II is the *Mathematical Tables Project*, funded by the *Works Project Administration* (WPA)[5] in 1938 as a reaction to the Great Depression. It was initially intended as a work relief project to create jobs for unemployed clerks and office workers, who should tabulate higher mathematical functions such as exponential functions, logarithms, and trigonometric functions. Due to the WPA policy, which required that all projects use the most labor-intensive methods available to keep the people busy [100], most of the calculation was done by hand, although calculating aids, such as slide rules or mechanical desk calculators, could have been available.

Detailed task preparation led to a breakthrough of human computers

However, under the technical direction of *Gertrude Blanch* (1897 - 1996), the Mathematical Tables Project succeeded as an organization of large-scale scientific computation. It became the largest[6], the most successful and most influential computing organization prior to the invention of the digital electronic computer [102]. Blanch pioneered the work of preparing detailed computation plans and breaking the computation down into smaller units that used only basic arithmetic operations. Hundreds of computers with little to no education could perform these basic operations in parallel, so people would usually perform only a single operation, such as addition or multiplication [101]. By introducing self-checking worksheets, the project resulted in highly reliable calculations and became the "most successful mathematical tables project in history" [254].

Soon after the publication of the first of 28 nearly error-free volumes of mathematical calculations, the project gained attention from the scientific community. It became involved in the calculation of several scientific as well as military problems until the end of World War II. One example is the computation of nuclear fission and shock wave propagation tables during the famous *Manhattan Project*, which was set up for research purposes in nuclear technology and to develop the first atomic bomb [254].

These examples show the great need for human computers until World War II. The particular role of women during that era can be summarized by a memorandum from the Computing Group Organization and Practices at the NACA, dated April 27th, 1942, which explains:

5 The WPA was an agency established by U.S. President Roosevelt to alleviate unemployment through public works.

6 Reaching its zenith in 1941, the Mathematical Tables Project employed 450 human computers. [100]

"It is felt that enough greater return is obtained by freeing the engineers from calculating detail to overcome any increased expenses in the computers' salaries. The engineers admit themselves that the girl computers do the work more rapidly and accurately than they would and also feel that their college and industrial experience is being wasted and thwarted by mere repetitive calculation." [159]

Hence, in times far from gender equality, women had to show exceptional motivation and commitment to work hard and gain social and scientific recognition. Gertrude Blanch and the Mathematical Tables Project are examples of this. Starting as a work relief project, it "proved to be a transitional institution in the history of computing, promoting mass scientific computation and developing the numeric methods that would eventually be used on electronic computers" [101].

To conclude this section, the history of human computers shows that a single person, no matter how talented he or she might be, could not easily compute complex problems all by himself/herself. Instead, it needed many people working together as a group and the division of labor to solve complicated tasks, ranging from calculating the complete orbit of a comet to ballistic calculations in war times. This division of labor, however, needed two things. First, superior intellectual capabilities were required for the careful and correct preparation of the tasks, and second, physical and mental stamina were required to remain defiant until the end of work. Altogether, human computers learned "how to divide their labors, how to work with hierarchical management, and how to devise standard computing procedures" [102].

2.1.2 *WW II - 1950s: Electronic Computation*

Human computation reached its zenith in the early 1940s during World War II. Up to this point, it became a substantial field by demonstrating the effectiveness of its work organization through large and successful projects. It even had its own journal called "Mathematical Tables and Other Aids to Computation," which contributed to a living scientific community [99].

Of course, the history of human computers went hand in hand with the development of calculation tools to facilitate their work. Over centuries, these tools were subject to continuous technological change and evolved from the abacus and simple mechanical calculators over analog computers, such as differential analyzers, to electromechanical tabulating machines using punched cards [220].

Punched-card tabulators could work much faster than humans, but this advantage was lost since humans had to spend several days

preparing the machine [254]. In addition, a large group of human computers could still keep up with a tabulator relatively well, as shown by a famous "showdown between man and machine" within the Manhattan Project. In this showdown arranged by *Richard Feynman*, human computers had to compete against a tabulating machine by performing calculations for the plutonium bomb. For at least two days, the human computers were able to keep up with the machine but could not sustain their fast pace on the third day, so the punched-card machine began to move ahead decisively since it did not need to rest and, of course, did not get tired.

Machines started to outperform human computers

2.1.2.1 *The Transfer of Computing from Human to Machine*

The challenge between humans and machines was of no means with the introduction of *electronic* machines towards the end of World War II, which clearly outperformed human computational power in speed and accuracy.

One of the most famous examples of these very first machines is the *ENIAC*[7], known as "the world's first large-scale-digital electronic general purpose computer" [37]. ENIAC was developed by *J. Presper Eckert* and *John W. Mauchly* at the *Moore School of Engineering* as part of an alliance between the University of Pennsylvania and the U.S. Army [289]. When finally revealed to the public, ENIAC was able to perform several thousand additions per second.

ENIAC

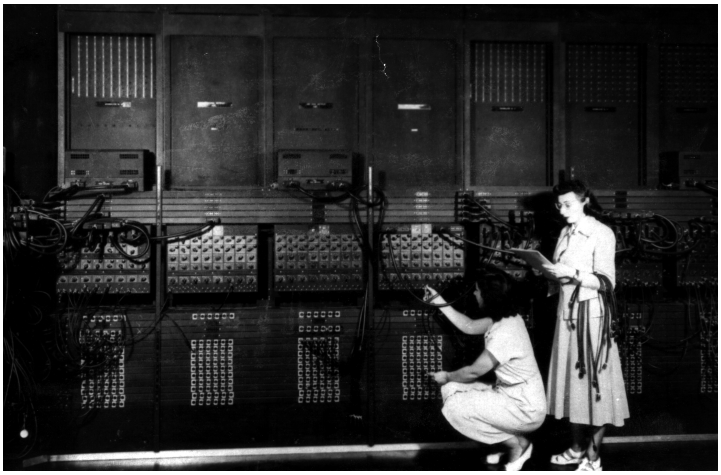


Figure 2.1: Two women wiring ENIAC[8]

The physical dimensions of this machine were nothing but huge. It filled an entire room of 5 x 10 meters, weighed 30 tons, and consisted of over 17.000 vacuum tubes and 70.000 resistors. For its operation, it relied on 6.000 manual switches within 40 electronic panels that were

7 ENIAC is short for "electronic numerical integrator and computer."
8 Source: https://ftp.arl.army.mil/ftp/historic-computers

arranged in a U-shape along three walls [167]. Also included were three mobile function table units and IBM punch card machines [185], while programming was done by plugging cables, thus defining the flow of electricity through its various components.

War times motivated the development of electronic computers

The motivation for the development of electronic computers during World War II was grounded in the calculation of ballistic tables for new weapons that were "terribly difficult to make" [12, p. 109] and required solutions to thousands of differential equations, which had always frustrated scientists since they need a massive amount of computations [167].

In this context, the ENIAC was intended to automate the production of firing tables for rockets and artillery shells, which formerly were calculated by nearly two hundred women [159] working as human computers at the *Ballistic Research Laboratory* (BRL) of the U.S. Army's Aberdeen Proving Ground.

While Eckert led the hardware engineering, Mauchly was head of ENIAC's conceptual design. Beforehand, he was an active member of the human computers' scientific community and lectured at the *Moore School of Engineering* about the existing principles of translating complex mathematical problems into a series of much simpler calculation steps making use of nothing but additions and subtractions [99]. He transferred these principles to ENIAC's design so that operators of the machine could still rely on existing techniques of problem decomposition [185].

2.1.2.2 *The Arise of "Programming"*

Humans became "operators" of electronic computers

Since Mauchly modeled ENIAC's computation techniques after those established by human computers, he also had to initiate a shift in job definitions [159]. Before ENIAC, a "computer" was a human being, but since ENIAC claimed the term for itself, humans then became "operators" and "programmers" [254] of the new machines.

In particular, six women formerly working as human computers at the BRL were selected to program ENIAC. Since none of them had ever programmed a computer before, they had to reverse engineer the machine and educate themselves by using only a bunch of logic block diagrams and creeping around the inside to understand how this machine could be operated [159].

Even worse, the ENIAC project incorporated a clear distinction between the engineering of hardware (a man's job) and its programming, which was treated as a "soft" and clerical task. As a result, women received little to no credit for their work [159]. However, this would change in later projects during the following years, when women

especially became pioneers in programming and also taught trainees on electronic computers [97].

What makes ENIAC particularly interesting in terms of computational history is the intended break from the era of human computers. Although Mauchly and Eckert treated electronic computation as an automated form of human computation using the same principles and techniques, none of the ENIAC inventors built up on the existing terminology used to describe organized computing, which was "plan" [99]. In comparison, *Konrad Zuse*, also a leading computer pioneer of this era, did theoretical work on the first higher programming language and made an apparent reference to the experience of human computation by calling it "Plankalkül" (plan calculus). However, the ENIAC team separated their work from that of human computers and therefore chose language accordingly [99].

In 1942, Mauchly introduced the term "to program" in his paper on electronic computing [172] but used it in the manner how ENIAC was configured, namely by the repetitive plugging of cables to define data flows across various units as well as signal flows telling each unit when to operate [185].

However, only four years later, in 1946, Eckert made the first use of "to program" in today's modern sense in one of his lectures at the Moore School of Engineering. This was preceded by the joining of *John von Neumann* to the ENIAC team in order to improve the static design of the ENIAC since it had to be re-cabled for every problem to be solved, which was cumbersome and limiting the machine's area of application [185]. As a result, they revised ENIAC into EDVAC[9] - the first computer incorporating von Neumann's concept of a stored program, nowadays known as "*von Neumann architecture*."

To conclude this section, the rise of electronic computation constituted a significant breakthrough. While earlier machines represented nothing more than the automation of techniques and principles developed by human computers, it can be seen, especially by the example of ENIAC, that computation has since undergone a change in paradigm.

Besides the aspects mentioned above, the impact of ENIAC on this shift can also be seen in particular by how it was unveiled to the public in 1946, which should have a dramatic impact on the public consciousness and perception of computers in general. This event was set up for maximum astonishment of the press and included the pompous announcement of adding 5.000 numbers together by a press of a button, which ENIAC accomplished within one second, even before the reporters had looked up, followed by a demonstration of

9 Electronic Discrete Variable Automatic Computer

calculating the trajectory of a shell, that was faster than the shell itself would need to hit the target [167].

As a result, the capabilities of electronic computation spread like wildfire all across the world. Newspapers made extensive use of vastly exaggerated metaphorical images describing these new machines as "electronic brains" and as if they were healers of modern civilization so that even Mauchly himself had to make clear that "the electronic calculator does not replace original human thinking, but rather frees scientific thought from the drudgery of lengthy calculating work" [167].

2.2 FROM THE ARISE OF SOFTWARE TO THE SOFTWARE CRISIS

At the beginning, "programming" was equal to physically wiring cables

Programming the early electronic computers was done by plugging cables to define the flow of data and the route of signals that control the corresponding units for data processing. This way of programming was very similar to switchboard operators working at telephone companies, where calls were connected by inserting a pair of phone plugs into the appropriate jacks of the board [97]. Similar to phone calls, which needed a change in the connection of cables for varying receivers, early computers needed a new physical setup and rewiring for every program to be run. Even worse, the programming setup highly depended on the physical structure of that particular machine.

Simply put, there was no distinction between hardware and software [268]. Instead, the "soft" part was represented by the operators themselves, and programming (a woman's job) was assumed to be of little difficulty and generally was valued as less meaningful than the engineering of hardware (a man's job).

There was neither a distinction between the producer of a machine and its users [268]. That is because early computers were specific-purpose devices built for particular tasks. As a result, both the producer and the user were closely involved in the construction process. "User" in this context represents hardware engineers. As written above, the programming of the actual machine was considered to be less critical, which is why operators (which we would associate with "users" today) had to deal with machines with almost no instructions or guides [97].

The memory concept of "stored-programs" enabled the real "programming" of computers

Overall, the main reason for these limitations of early computers was lying in a missing memory. Hence, the now famous introduction of stored-program computers by John von Neumann [186] and the ENIAC team, as explained in the previous section, constituted a breakthrough in computing technology, proving ground for upcoming digital instructions to the computer and getting rid of physical wiring.

2.2.1 *1950s - The Arise of Software Development*

The 1950s can be described as the transition period in which the paradigm of computer operation was subject to change and went from "wiring" of the 1940s over "coding" in the 1950s to "programming" in the 1960s. However, this decade was still characterized by a predominant position of hardware engineering orientation[10].

While hardware engineering made steady progress, software aspects proceeded slowly. Computers were mainly used by governments for military purposes and treated as "computing machines" [67] or automata for extensive mathematical calculations, which must be "coded" into algorithms that the machine could process. This "coding" involved a high amount of analytical and intellectual power since it remained very close to higher mathematics, such as numerical analysis [67].

Computers had high operation costs and were mainly used for military purposes

Overall, the area of computer application seemed very limited, not least because of the high expenses for operation, which is nicely illustrated by a quote from Barry Boehm, who worked at the defense company General Dynamics and remembered the instruction of his supervisor at his first day in the job in 1955:

"Now listen. We are paying $600 an hour for this computer and $2 an hour for you, and I want you to act accordingly." [32]

The UNIVAC opened electronic computation to general businesses

Consequently, this led to programmers following the advice of "measure twice, cut once" by doing a tremendous amount of testing and manual execution of programs before running them on a computer [32]. However, there were also rising efforts to widen the computation field and establish computers for performing practical tasks, such as payrolls and inventory [107]. The clearest example can be seen in the *Eckert-Mauchly Computer Corporation*, founded by the ENIAC inventors, who were convinced that computers could be used for universal tasks in the economy instead of just mathematical problems. Their *UNIVAC* was the first commercially available computer in the United States and allowed them to perform universal data processing [211], which opened the field of computation from military to general business applications.

Of course, the intent to sell computers commercially meant not only to make them perform business tasks in any manner but also to enable users to program these machines according to their needs. For this reason, the Eckert-Mauchly Corporation also impacted the rise of the first programming languages. Similar to the era of human computers,

10 This predominant orientation towards hardware engineering still shows in the names of today's leading scientific communities for software professionals: the ACM - Association for Computing *Machinery* and the IEEE *Computer* Society.

it was again women who pioneered the rise of programming electronic computers and making them accessible to society [107].

The rise of programming languages: getting computers to perform practical tasks

In this context, it is essential to mention *Grace Murray Hopper* (1906 - 1992) and *Betty Holberton* (1917 - 2001), both working on the UNIVAC project. Hopper came up with the idea of using a higher language to program computers instead of bare "0" and "1" instructions. Some of her outstanding achievements were the introduction of the first compiler named *A-0* and the development of *FLOW-MATIC* [143] as a direct predecessor of *COBOL*, a programming language strongly modeled on natural language and intended for business applications, which is still in use today.

While Hopper stood out regarding visionary concepts of hardware-independent and comprehensive programming languages, Holberton excelled in her exceptional programming work. Beforehand, she was one of the six human computers selected for the "programming" of ENIAC [159]. During the UNIVAC project, Holberton developed the first program to sort and merge files, which solved a key problem for business applications [107]. In later years, she contributed to the enhancement of *FORTRAN* - the first high-level programming language. Her remarkable work can also be seen by the deep appreciation of her colleagues since Hopper described Holberton as "the best computer programmer she knew" [107] and one of her former ENIAC workmates said that "Betty could do more logical reasoning while she was asleep than most people can do awake" [89].

2.2.2 *The mid 1960s and the Software Crisis*

Since the 1950s, computers have started to leave closed laboratories and set foot in the business world as a data processing tool. Times when these machines were accessible to only a few insiders very soon belonged to the past, and from then on, they became part of the public domain [290]. This business domain introduced new challenges. It needed solutions to search, sort, and manage large amounts of data [220] so that the development of software (in the form of little programs) began to separate from the domain of hardware engineering.

Computers became part of the public domain and caused a rapidly growing demand for software

As a result, computers reached a new stage during the 1960s and were used as information systems [67]. The development of memory mechanisms now made it possible to handle large sets of data, so computers were increasingly used to automate administrative work on the one hand and to use them for monitoring tasks of industrial production processes on the other hand.

Overall, there was a rapidly growing demand for new systems and application software, which far exceeded the availability, so pro-

grammers were increasingly falling behind [220]. On the other hand, research made massive progress during the 1960s. Computer science was established as a new discipline at universities, and the number of informatics departments started to grow, now with increasing emphasis on software [32].

The tremendous impact of that era is still visible today. Within approximately just one decade, software has grown from basic programs for sorting and merging files to the first commercially selling products delivered by software development companies. Programming languages have evolved from first assembly to higher-order languages, such as Fortran and COBOL. Furthermore, computer science - just founded as an individual discipline - demonstrated groundbreaking innovations ranging from the introduction of computer graphics and artificial intelligence [220] to the spectacular "Mother of all demos" by *Douglas Carl Engelbart* (1925 - 2013) at the Fall Joint Computer Conference in 1968. In this demonstration, Engelbart showed almost all elements of today's personal computing, like windows, the computer mouse, hypertext, video conferencing, and collaborative work using a real-time editor, and so basically founded the research field of *Human-Computer Interaction* (HCI), including the upper goal to use computers for the "augmentation of the human intellect" [81].

From the very beginning, people noticed that the development of software was utterly different than the engineering of hardware. It was much easier to modify the code once and reload new copies to different computers afterward. In contrast, a change in hardware configuration meant changing each individual instance of a system. Due to this ease of modification, the programming community adopted a "code-and-fix" approach representing the exact opposite of the principle of "measure once - cut twice," which was the prevailing paradigm for hardware engineering [32].

The need to deliver software quickly led to software crafting and "spaghetti code"

This "quick and dirty" delivery of products became a primary concern because the rapid expansion of demand for software and business applications outstripped the supply of programmers (who were mathematicians at that time). In turn, this made software companies open their doors to people from other different disciplines so that non-mathematicians could enter the field of programming and "flooded into software development" [32].

However, both aspects, the approach to code first and fix afterward and opening the software development process to people of other fields, led to "spaghetti code" that was merely cobbled together and hence extremely difficult to maintain. As a remark, this situation was also aggravated since many of these non-specialists were influenced by the well-known zeitgeist of the 1960s and the attitude to question authorities, which conflicted with the companies' goal to meet project deadlines [32].

As a result, software applications became very people-intensive, and estimating the costs of a project was more by the rule of thumb than profound business models [163]. *Winston Royce* nicely illustrates this in his paper "Managing the Development of Large Software Systems," in which he said:

"In order to procure a $5 million hardware device, I would expect a 30-page specification would provide adequate detail to control the procurement. In order to procure $5 million worth of software, a 1500 page specification is about right in order to achieve comparable control." [224]

The software crisis caused many failing development projects

To sum up, the 1960s were an era in which software development was evolving rapidly. Projects became increasingly difficult to estimate in terms of costs and scope. In addition, reliability models only existed for hardware projects and could not simply be transferred to the software domain. People had to learn that developing software is a much different activity than developing hardware:

"Software was invisible, it didn't weigh anything, but it cost a lot. It was hard to tell whether it was on schedule or not, and if you added more people to bring it back on schedule, it just got later." — Barry Boehm [32]

This situation, known as the *software crisis*, led to many failing projects, either in terms of budget or schedule (or both).

2.3 SOFTWARE DEVELOPMENT AS AN ENGINEERING DISCIPLINE

As written in the previous section, the turning point at which software separated from hardware was during the 1950s and was established by technological progress in terms of hardware (mainly memory mechanisms) and, in particular, by the rise of the first compilers and programming languages. This separation did not happen overnight but instead was an ongoing development, which is why a precise hour of birth of software cannot be given, as this also depends on how "software" is defined.

Riding on the wave of technological innovation, it was during the 1960s when computers left laboratories and entered the business and industry sectors, leading to an exploding demand for software applications. However, projects could only be estimated and scheduled on a rule-of-thumb basis because of missing techniques for proper process management. In addition, software has been implemented by a "hacking" approach through programmers with a very heterogeneous skill set and work performance. Overall, projects encountered a wide range of problems and failed on a regular basis.

2.3.1 *1968 - The Arise of "Software Engineering"*

Striving for solutions, the *NATO Science Committee* sponsored two landmark conferences in 1968 (Garmisch, Germany) and 1969 (Rome, Italy). While the second conference is of minor concern and was "far less harmonious and successful than the first" [214], the Garmisch conference had a tremendous impact on computer science in general and on the development of software in particular.

The NATO Conferences started to establish software development as an engineering discipline

It was chaired by *Friedrich Bauer* and attended by more than 50 people from 11 countries, all professionally involved in the software field, including many of the leading researchers and practitioners [40]. Their common goal was to work together against the existing problems of late deliveries of software having low reliability but high maintenance costs - or what they referenced by the term "software crisis."

As a starting point, Bauer coined the term *"Software Engineering"* and chose it as the conference title, which was "deliberately provocative" [214], but already indicated the direction in which software development should be heading. Namely to better methodologies and tools as well as "more disciplined practices" [32] in order to strive for better software quality control and minimize the likelihood of failures [92].

Although critical comments about the programming profession had been stated in the years before, it was not until the NATO conference that the term "software crisis" was coined and that problems and difficulties had been discussed very openly and "with unusual frankness" [290]. The list of topics was long and covered the whole spectrum ranging from the relation of software to computer hardware, the design, planning, and implementation of software, to its distribution and maintenance after delivery [40].

However, as Broy mentioned in 2018 while reflecting on the last 50 years of software engineering that followed the Garmisch conference, it was not only the list of topics that paved the way for subsequent research but also what was initiated by the newly introduced term "software engineering."

By defining the development of software as an engineering discipline, people became aware of the challenges, for instance, the necessity to establish a scientific foundation and base all upcoming methods, techniques, and tools on scientific theories, which can be validated and proven in terms of correctness and efficiency (since this is characteristic for an engineering discipline). As a result of this conference, many of the early contributions to the field of software engineering were questioning the existing, and the community started to intensely focus on a stepwise implementation of a scientific foundation, which then

opened the stage for many innovations, of which some are covered in the following section.

2.3.2 *Reflecting on 50 Years of Software Engineering*

> *"Our discipline has exploded since 1968. In 1968, there were a number of key people who seemed to understand the whole discipline in all its facets. Since then, our field has come up with so many different subdisciplines and areas of application that no one can grasp the whole field [...] anymore."*
>
> — Manfred Broy [40]

Controversy about the term "software engineering"

As a first step, what is meant by the term *software engineering*? It seems that even this simple question has led to many discussions about the term itself, which started right after its introduction in 1986 and still lasts to this very day. From the introduction to the proceedings of the first NATO conference, it says:

> *"The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."* [214]

Nearly twenty years after its introduction, *Michael Mahoney* called attention to the controversy about the term in 2004 by at first quoting a leading practitioner, who in 1989 defined "software engineering" as "the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software" [163]. This is followed by another quote from a colleague of this practitioner, who declared in 1990 that "software engineering is not yet a proper engineering discipline, but it has the potential to become one." By these examples, Mahoney referred back to the excerpt above and stated that this phrase was indeed provocative, but only because all crucial terms were left undefined:

> *"What does it mean to 'manufacture' software? Is that a goal or current practice? What, precisely, are the 'theoretical foundations and practical disciplines' that underpin the 'established branches of engineering'? What roles did they play in the formation of the engineering disciplines?"* [163]

He then argues that "every definition of software engineering presupposes some historical model" and explains the controversy about "software engineering" by saying that the participants of the first two NATO conferences influenced the field up to the present time by their varying professional and disciplinary traditions. These traditions,

in turn, result from their historical background and views of "engineering," which could be applied science, mechanical engineering, or industrial engineering [163].

According to Mahoney, the focus on "engineering" was more on the side of applied science for the first years after the two NATO conferences, which in particular meant theoretical and mostly mathematical computer science. However, during the mid-1970s, it should have transpired that mathematical models could not suit the diversity of computing and would not quite meet the needs of software engineering. That is why the interpretation of "software engineering" shifted towards the influence of the other two domains.

He illustrates this by associating the upcoming concepts of "modularity" and reusable code routines during the 1970s, as well as the resulting orientation towards achieving such reusability on a broad scale by object-oriented programming during the 1980s, to models of mechanical and industrial engineering, including their principles, such as "interchangeable parts," "division of tasks," and "mass production."

Further, he elaborates that under the influence of industrial engineering, the objective of software engineering became "automation," which may be seen in rising terms within the scientific community, such as "software factories" and the comparison of software development procedures to industrial assembly lines. However, he concludes that a factory-like form of engineering focusing on automation will not solve software engineering problems and will lead to the misinterpretation of the field. Finally, he questions the term by saying that "practitioners disagree on what software engineering is, although most of them freely confess that, whatever it is, it is not (yet) an engineering discipline" [163].

This argumentation is in line with *Antony Bryant*, who considered the impact and role of the "engineering" metaphor and argued that metaphors play an influential role "as an indispensable component of cognition." He concludes that the engineering metaphor, in particular, has moved us forward during the "critical activity of developing a discipline for software development," but that "we now have to move forward to the next stage" [41].

While the term and its implications for the field are still controversial within the scientific community, there is no question about the actual achievements and innovations of software engineering during the last 50 years.

The following briefly touches on some examples intended to give a small impression of the incredible variety of the associated research field and to provide a broader perspective on the subject matter of this thesis.

Software engineering led to new ideas for more careful programming

After the NATO conferences and the birth of software engineering as a new discipline, there was a new momentum to overcome the existent "code-and-fix" approach and to spend more time on a *careful analysis of the coding activity* [32]. As an example, Boehm points out that a letter from Edsger Dijkstra [75] has led to the movement of *structured programming*, which in turn paved the way for concepts such as "modularity" of code, "information hiding," "abstract data types" and finally led to *procedural programming* techniques, such as the revolutionary *Pascal* by *Niklaus Wirth* [290] and todays modern *object-oriented programming languages*.

Besides better programming concepts and languages, there have also been many achievements in terms of *achieving better code quality*. Today, powerful IDEs[11] and incorporated tools help programmers write code according to style guides and patterns to ensure a certain level of code quality. In addition, comprehensive *software testing methods* have been developed to guarantee error-free operation for at least parts of the software[12].

Maybe the exact opposite to the 1960s approach of code first and fix afterward can be seen in the subdiscipline of *model-driven software development*, which started during the 1980s and is still the subject of many research projects today. This software development methodology focuses on abstract representations of knowledge (models) as the primary artifact of the development process so that the implementation of code can be (semi)-automatically generated from the models.

However, for this thesis, what has been achieved in terms of *managing the software development process* is of particular importance. In 1976, Boehm defined software engineering as "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them" [26]. This definition shifted the focus of software engineering to what should become the *standard lifecycle model* of software, which is today's understanding of the software development process as an interplay between the specification of requirements, design considerations, the coding activity and its verification through testing, as well as aspects of delivery, operation, and maintenance of the product.

The next chapter will present the evolution of software development lifecycle models and explain why agile methodologies and Scrum became the leading paradigm. Beforehand, the following section extracts some "lessons learned" from this chapter about the historical background of software development.

11 Integrated Development Environment

12 Testing cannot detect all errors in a program as it is impossible to assess every execution path except in the simplest programs. This issue encompasses the *halting problem*, which is inherently undecidable.

2.4 SUMMARY AND LESSONS LEARNED

Of course, analyzing the historical background and evolution of software development has many facets and could quickly fill several books. Therefore, presenting only a rough sketch, as in this chapter, will naturally leave important things out and may not satisfy interested readers. However, this chapter is by no means intended to tell all the exciting stories of all those remarkable men and women who paved the way and contributed to how software is developed today.

Instead, it is more about giving an impression of the manifoldness and complexity of the software development procedure based on some historical landmarks and turning points, delivering some "lessons learned" for the present time, and trying to build a bridge to today's agile way of developing software using the Scrum framework, which is the focus point of this thesis and will be presented in Chapter 4.

The value of teamwork over individual genius

These lessons reach as far back as the middle of the 17th century, when "computation" was pure mathematics and usually performed by highly educated experts working independently behind closed doors. However, as seen by the example of Comet Halley (see Page 10), collective efforts often surpass the capabilities of individual experts. This evolution from isolation to collaboration is a cornerstone of modern agile methodologies like Scrum, where *communication, teamwork, and the sharing of ideas* is prioritized over individual prowess. The transformation of "computation" from a solitary mathematical activity to a collaborative, multifaceted process mirrors the evolution in software development. It underscores the necessity of adapting to changing definitions and scopes of work, just as software development has evolved from a technical coding task to a complex, collaborative process.

The importance of careful and correct task preparation

At the beginning of the 19th century, a turning point in the era of human computers was the *division of labor*, initiated by Gaspard de Prony when calculating trigonometric and logarithmic tables (see Page 11), which was later brought to perfection by Gertrude Blanch during World War II in the Mathematical Tables Project (see Page 13). What can be derived from these examples is the utmost importance of *careful and correct preparation of tasks* prior to the actual work execution. This insight can be easily connected to today's development practice because, as will be shown in later chapters, many failing software projects show problems in the implementation or coding phase (work execution) as a result of failures in the specifications of requirements.

Moreover, the division of labor at the beginning of the 19th century resulted in an intellectual shift and "factory-style" of (human) computing (see Page 12), which interestingly shows a connection to the 1980s, when the term "software factory" was coined. At that time, software development was compared to industrial assembly lines [163]

Understanding "programming" as a creative process

and similar to Prony's division of labor and letting unskilled people perform myriads of simple calculations, the movement of "automatic programming" and "computer-aided software engineering" (CASE) had the ultimate goal of taking a problem specification and letting the computer transform it automatically into a working system, which essentially means eliminating the programmer, just like Prony wanted to eliminate the need for skillful mathematicians. However, in contrast to the computation of Prony's mathematical tables, programming turned out to be a *thoughtful and creative process* so that the idea of automatic programming ultimately failed with programmers complaining about the term "software factory" since it connotes a devaluation of their skills [65].

The need to establish cross-cultural awareness

History has shown how electronic computers initiated a paradigm shift in computation and superseded the work of human computers, which then became operators of these machines. The evolution of this "operation," starting from "plugging" cables, over "coding" in machine language to finally "programming," illustrates the liberation of the mind from dull and repetitive work towards creative thinking. The crucial, yet often overlooked, contributions of women and their pivotal role, especially in the transition from manual to electronic computation (see Page 20), highlight the need to establish cross-cultural awareness and integrate diverse perspectives and talents into the software engineering discipline. As will be shown in Chapter 4, Scrum builds up on this insight and proposes that understanding and valuing diverse influences can lead to more innovative and effective approaches in software development.

The separation from hardware and the development of "programming" marked the beginning of the triumph of software, which is now everywhere and will continue to be ubiquitous in the future. At the same time, the software crisis has revealed the inherent challenges of software engineering, as there is nothing that can be quickly adapted from other disciplines, leaving people in disbelief as to whether software engineering is an engineering discipline at all.

People have been looking for solutions to overcome the software crisis for fifty years. Researchers aimed to improve product productivity and quality and developed new programming languages, test mechanisms, and formal methods. In addition, new process models have been proposed to better manage all aspects of the software development lifecycle [92], as shown in the following chapter.

Nevertheless, despite all these efforts, projects still fail on a regular basis in terms of budget, scope, or both of them. None of the achievements within the software engineering community represented an ultimate solution, and as Brooks has shown in his famous essay "No Silver Bullet: Essence and Accident in Software Engineering" [38], there may not be a "silver bullet" to all the existing problems.

Agile software development approaches as an answer to the software crisis

However, as will be shown in the next chapter, *agile* software development approaches that emerged approximately twenty years ago initiated a paradigm shift by incorporating a completely different philosophy of how software should be developed, thus bringing programming, which is now treated as a highly collaborative team effort, into a new era.

Before Chapter 4 presents Scrum as today's most prominent agile approach, and the subsequent course of this thesis will sharpen towards the central subject matter, Chapter 3 first elaborates on the evolution of lifecycle models. Again, this is intended to provide enough backdrop and historical context for the later analysis of Scrum's issues and challenges, which will finally be addressed by a novel solution designed for optimal support of Scrum teams and their specific requirements.

3

THE EVOLUTION OF SDLC MODELS

"We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty."

— Donald E. Knuth. "Computer Programming as an Art." [141]

As explained in the previous chapter, the term "software engineering" was coined after the first NATO conference in 1968, and the newly established discipline gathered momentum in the search for solutions to problems of the existing software crisis.

One aspect was better management of the overall process and establishing a model for the different software development phases, known as the software development lifecycle (SDLC). During the last fifty years, numerous approaches and methodologies for implementing this lifecycle have been introduced, each focusing on particular aspects. Since it is far out of scope for this thesis to elaborate on all of them, this chapter will introduce only a selection of models and methodologies that have proven to be most relevant for the scientific community and which gained more momentum than others, thus contributed a lot to the overall evolution towards today's agile development philosophy.

Since this thesis introduces a novel solution to the development practice and the very specific requirements of agile Scrum teams, Chapter 4 elaborates on all details of Scrum, whereas the present chapter is mainly intended to deliver some historical context and insights from different models that preceded this development framework. Furthermore, these previous development principles provide context for understanding the research questions of this thesis, which will be explained in Chapter 5, and the subsequent analysis of challenges when implementing Scrum, which will be covered in Chapter 6.

First, a definition of the standard model of the software development lifecycle and a subsequent section clarifying terms should help the reader avoid getting confused when stumbling over different terminologies, such as "SDLC model," "SDLC method," "SDLC methodology," and so on. In addition, a short section about how SDLC models could be classified should assist further reading by differentiating between different types of models.

3.1 SDLC DEFINITION

The *software development lifecycle* (SDLC) can be defined as the sum of various phases or activities within the overall process to genuinely build a software product from the ground up (see Figure 3.1).

Figure 3.1: The software development lifecycle (SDLC)

While Figure 3.1 shows seven phases, other representations of the software development lifecycle may only have six or five. This is usually when the seven phases are described in technical aspects instead of higher-level activities. For example, the "planning" and "analysis" phases are often reduced to "requirements analysis." Likewise, "deliver" and "maintain" are sometimes reduced to "deploy and operate." However, this difference is just a question of granularity and does not affect the distinction between the lifecycle's individual phases.

1. *Planning*

   Every software project involves different kinds of planning activities. On a technical level, these activities include specifying requirements and determining what the system should accomplish. On a higher level, planning also defines a roadmap for the software project based on its scope and budget.

2. *Analysis*

   The analysis phase is a buffer between gathering and acting on requirements. It involves client approval or clarification about the project to ensure the right track before more detailed work is started. On a technical level, this phase defines the structure of system requirements to understand all aspects of the starting position and an eventually existing system that should be improved.

3. *Design*

   The design phase considers *how* the product will fulfill the previously identified user requirements. It involves making many technical decisions and creating detailed descriptions, plans, documents, and specifications about the software to be built.

4. *Coding*

   In the coding phase (also known as the "development" or "implementation" phase), the theory is turned into practice, and all preceding design decisions are transformed into working program code.

5. *Test*

   The testing phase is meant to ensure the correct behavior of the implemented requirements and to satisfy a certain level of code quality. During the development of complex systems, which involves not one but several or many developers, parts of the program code will likely be modified or overwritten by accident. Different test levels attempt to execute different program parts to find errors or other defects. *Unit tests*, for example, verify the functionality of a specific section of code, whereas *iteration tests* expose defects in the interplay and interaction between integrated components (modules). In contrast, *system tests* run on a completely integrated system to verify that the system meets the user requirements.

6. *Deliver*

   In the deliver phase (also known as the "deployment" or "operation" phase), the built software is delivered to the customer, integrated and installed into the existing system environment, and put into operation.

7. *Maintain*

   The final stage of a software development lifecycle occurs once the built product is fully operational. Maintenance involves monitoring the system performance and rectifying still-occurring software bugs, managing change requests, and keeping the system live with necessary updates.

At first glance, these phases define a logical, sequential order based on the temporal relations between the phases. For instance, no one would assume a system could be maintained before it is developed. However, the same does not apply to testing, as an example, which actually *can* be done upfront of a succeeding development or coding phase (by an approach called test-driven development).

Hence, although a particular order of these phases seems natural and reasonable, this must not be the case. In fact, this insight goes hand in hand with the historical evolution of SDLC models and the shift in paradigm from sequential development processes to agile methodologies, as will be illustrated later in this chapter.

Beforehand, and to avoid any misunderstanding, the terms "model," "methodology," and "framework" are examined more closely and demarcated against each other since they will be frequently used in the further course of this thesis.

3.2 TERMINOLOGY: MODEL VS. METHODOLOGY VS. FRAMEWORK

Models

A *model* is an abstraction or simplified version of an aspect of the real world. Therefore, an *SDLC model* is an abstract representation of the software development lifecycle. In contrast, a process *methodology* is a concrete manner of how software is developed.

Methodologies

That means an SDLC model does not specify how to do things but only outlines the types of things done during the lifecycle. In contrast, a methodology is a specific way of conducting a software project and precisely defines what, when, and how various artifacts are produced. In that sense, a model is *descriptive*, whereas a methodology is *prescriptive*.

Frameworks

However, some methodologies might only be partially explicit with all regards and are therefore called *frameworks*. For instance, Scrum, the agile methodology of primary concern for this thesis, is considered a framework since, although it is peculiar in many aspects, it also leaves lots of freedom to those who apply it.

The later part of this chapter will show that software development processes have shifted from traditional SDLC models to the agile SDLC model, which encompasses many *different* agile methodologies. Moreover, before this paradigm shift occurred, speaking about representations of the development process was focused on "models." In that sense, the evolution of software development processes can be seen as an emergence of abstract process descriptions (models) to concrete process designs (agile methodologies).

3.3 CLASSIFICATION OF SDLC MODELS

Similar to the differentiation between "model," "methodology," and "framework," it makes sense for a better understanding of the course of this chapter to quickly investigate the set of attributes or (in some

cases) synonyms the reader might stumble upon when various authors characterize different kinds of SDLC models.

For example, Ruparelia stated in 2010 that SDLC models fall under three categories: *linear*, *iterative*, or a *combination of linear and iterative* models [227]. However, this classification must be considered incomplete because it leaves out *incremental* models, leading to difficulties with classifying the agile SDLC model (since this is undoubtedly both iterative *and* incremental, as will be explained in the Sections 3.5 and 3.6).

In addition, various authors describe particular SDLC models with different attributes depending on the context and what they try to express. Therefore, a brief explanation of the terms commonly used to describe SDLC models (and methodologies) should help to prevent confusion and is intended to assist readers in further studies. The set of attributes for differentiation between SDLC models and methodologies is shown in Figure 3.2.

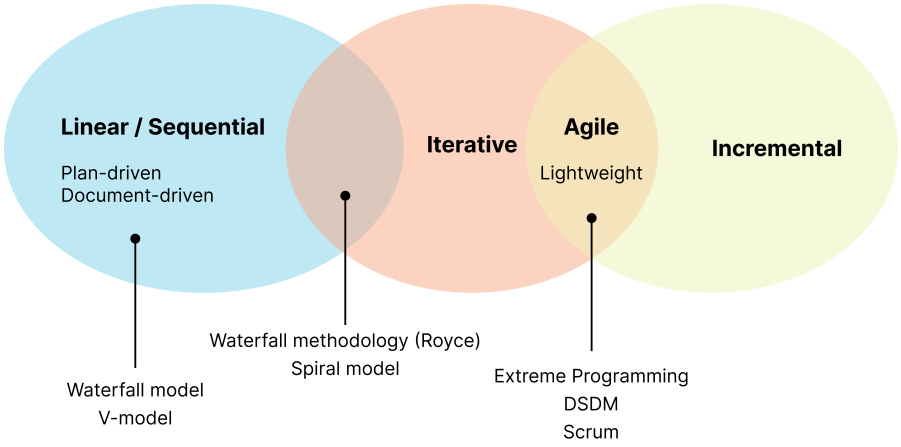


Figure 3.2: Classification of SDLC models

Linear models are also known as sequential, plan-driven, or document-driven

Linear, sometimes called *sequential* models, define the software development process as a series of stages, where the completion of a stage irrevocably leads to the initiation of the next one. As a result, the development process is strictly one-directional and must be planned and conducted with extreme care since there is no turning back and decisions of preceding stages cannot be revised. These models are also attributed as *plan-driven* or *document-driven* due to the proposed heavy amount of documentation and precise specification plans needed to complete a stage and enter the next one. Examples of sequential models include the *Waterfall Model* (see Section 3.4.1) and the *V-Model* (see Section 3.4.2).

On the other hand, an SDLC model described as being *iterative* means that a defined development process or concept is repeated so

Iterative models

that every phase is revisited in future iterations. This iterative process assumes that the artifacts or outcomes of any phase cannot be correct on the first attempt and that requirements may change over time so that development remains a constant endeavor for improvement throughout the lifecycle. Examples of iterative development include the modified *Waterfall methodology* by Royce (see Section 3.4.1) and the *Spiral model* (see Section 3.4.3).

Incremental models

An *incremental* model is characterized by a piecewise development and delivery of the product. So instead of developing all aspects of the product as a whole and over the whole time, resulting in finally delivering it in a "big-bang" style as one final piece, incremental development is about first splitting up the system functionality into numerous parts that are built independently from each other and second, delivering these parts as portions throughout the project. In doing so, the delivered parts build up on each other, and the product can grow as more "increments" add up.

The Agile model combines iterative and incremental development

The *Agile* SDLC model is a notable example of both iterative and incremental approaches, and its principles are reflected in all agile methodologies. Developed and refined since the early 1990s [52], agile software development methods mark a significant departure from pre-existing approaches and represent a paradigm shift in software development. The attribute "lightweight" describes the differences between agile methodologies and previous "traditional" approaches, with the latter being characterized as "heavy" in planning and documentation. In contrast, agile methods emphasize flexible and adaptive techniques over rigid processes and artifacts. Examples of agile methodologies include the *Dynamic Systems Development Method* (see Section 3.6.2), *Extreme Programming* (see Section 3.6.3), and *Scrum* (see Chapter 4), which is the primary concern of this thesis. Further elaborations on the paradigm shift from traditional to agile development principles will be given in Section 3.5.2, while the following sections will provide an overview of the evolution of SDLC models and their most prominent representatives.

3.4 PLAN-DRIVEN DEVELOPMENT

The evolution of SDLC models begins with plan-driven development processes, which, as explained before, are characterized by a heavy amount of upfront planning and documentation, followed by linear distinct development phases.

3.4.1 *Waterfall Model*

The most basic and earliest SDLC model is known as the *Waterfall Model* proposing a sequential execution of the lifecycle phases, as introduced in Section 3.1. Moreover, these phases are strictly separated, meaning that the work of one phase must be entirely finished before activities of the succeeding phase can start. In that sense, software development relates to a flow of water directed in one direction only, as illustrated in Figure 3.3.

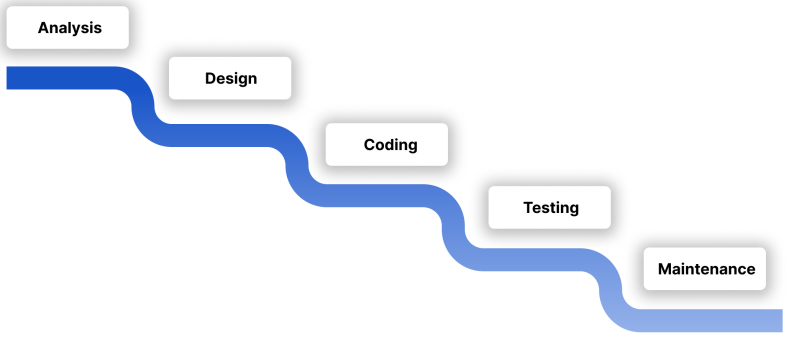


Figure 3.3: Waterfall Model

While the image shows five distinct phases, there are also variations with up to seven phases (for instance, by splitting the singular "Design" or "Testing" phases into two), but this is (as already stated in the explanations of the individual phases, see Section 3.1) more or less just a question of granularity and does not affect the model's concept of strictly separated and sequential phases.

Historically, the roots of the Waterfall Model can be traced back to 1956, when *Herbert Benington* documented a defined software development process for the *SAGE*[1] project for the U.S. and Canadian air defense [22]. While the SAGE model was specific for military purposes, it needs a small amount of abstraction or adjustment of wording to identify a relatively close match between the stages used by Benington and the phases of the later Waterfall Model [227] (see Figure 3.4).

The Waterfall Model stems from adapting existing hardware engineering phases to the development of software

Against the historical backdrop of software engineering, as described in the previous chapter, it is now clear that this very first SDLC model originates in the manufacturing process of hardware, in which changes to a physical product in later phases become prohibitively costly, if not impossible. However, since no formal software development methodology existed at that time, it seemed appropriate to adapt the hardware engineering model to software development.

The term "waterfall" was coined in a paper by Bell and Thayer in 1976 [21]. It led to some confusion since they referenced and asso-

1 Semi-Automated Ground Environment

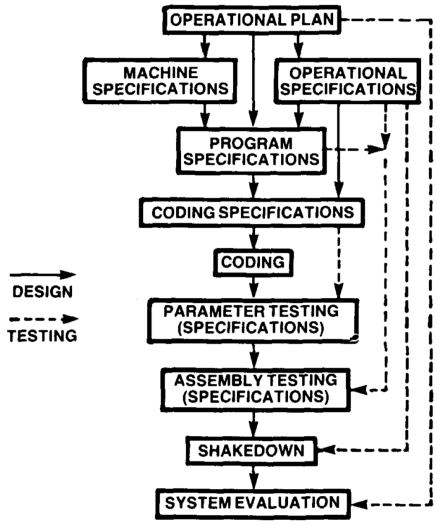


Figure 3.4: The first documented sequential process model [22]

ciated this term with a nowadays famous paper by Royce, who is often supposed to be the first person who formally proposed the Waterfall Model. However, neither Benington nor Royce has actually used that term. In fact, Royce was the first to announce the sequential model to a broader audience using the simplified form we are used to nowadays. However, he has also been the first to argue *against* the one-directional flow and strictly sequential development process and, therefore, proposed a version of the "waterfall" that is enhanced by various aspects.

At first, he identified the testing phase, which occurs at the end of the development phase, as the moment in time when the actual behavior of a system can be experienced, which is often very different from what has been analyzed and designed in preceding phases. This mismatch of expectations and reality could only be solved by a significant redesign, which may be "so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated" [224]. As a result, the development process would have returned right to its beginning, thus leading to a 100% overrun in schedule and costs (see Figure 3.5).

His solution to eliminate most of these development risks consisted of five additional steps, which are all aggregated into his proposed methodology[2], as shown in Figure 3.6.

As a first step, he inserted a (1) preliminary design phase between the software requirements generation phase and the analysis phase. This is intended to (2) initiate an early simulation of the final product

2 Notice the difference between the terms "model" and "methodology," as described in Section 3.2. Whereas the Waterfall Model is rather abstract, the proposed version of Royce is a precise process description, hence a methodology.

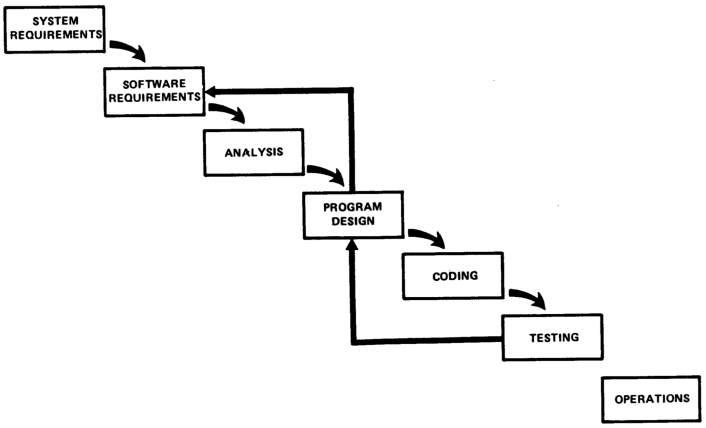


Figure 3.5: Restarting the waterfall because of testing results [224]

by what he called a "do it twice" approach and what would be called "prototyping" today. Furthermore, he insisted on (3) a heavy amount of documentation through all phases of the project, (4) enhanced testing by planning, measuring, and monitoring tests to guarantee proper behavior and what is interesting, (5) involving the customer, so that he has to commit himself at earlier points in time before the final delivery.

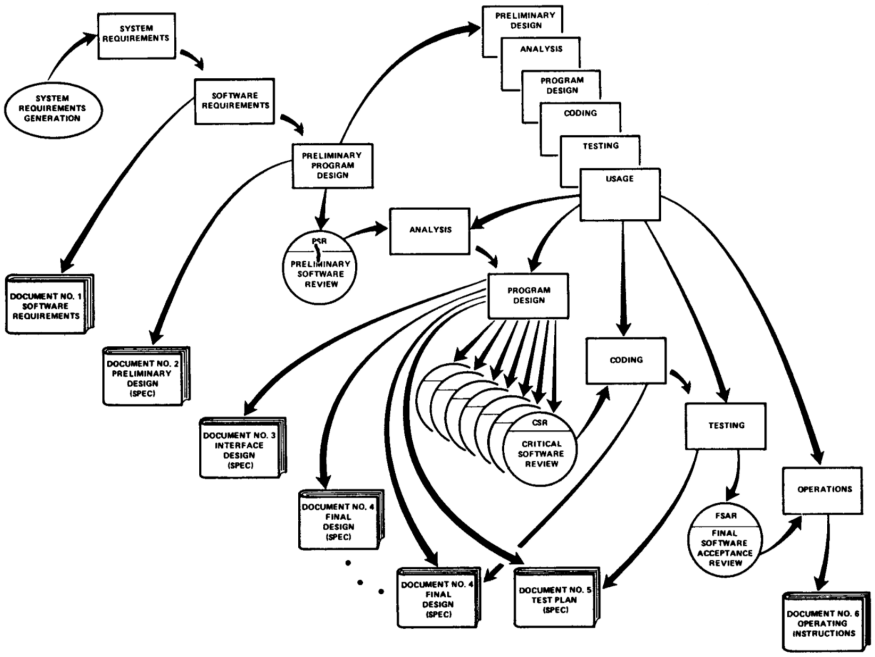


Figure 3.6: Proposed development process by Royce [224]

In the end, it is interesting to see that Royce is often falsely referenced as the originator of the Waterfall Model, although he actually identified several problems of this rigid sequential approach and proposed the solutions mentioned above. However, the reasons can-

not be reconstructed, and it can only be guessed whether his ideas have not gone far enough to circumvent this reference since many characteristics of the Waterfall Model are also valid for his proposed methodology.

3.4.2 *V-Model*

In 1979, Barry Boehm introduced the V-Model in his "Guidelines for Verifying and Validating Software Requirements and Design Specifications" [27]. From his studies indicating that the costs to find and fix software errors will exponentially grow while progressing towards later stages of the lifecycle (see Figure 3.7), he concluded the utmost importance of resolving software problems and high-risk issues of the lifecycle at the earliest stage possible.

"[S]avings of up to 100:1 are possible by finding and fixing problems early rather than late in the life-cycle. Besides the major cost savings, there are also significant payoffs in improved reliability, maintainability, and human engineering of the resulting software product." — Barry Boehm [29]

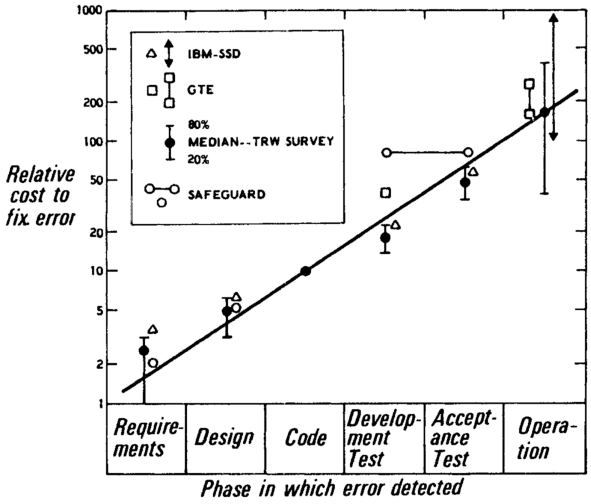


Figure 3.7: Relative costs to fix software errors [27]

Therefore, a central aspect of his V-shaped sequential process model was the emphasis on *validating* and *verifying* the results, as seen in Figure 3.8.

Software verification vs. validation

While *software verification* means to check whether the artifacts of a development phase, for example, program code and design documents, satisfy the requirements specified before, *software validation* considers if these artifacts meet the actual user needs. Boehm explains the difference very descriptively by treating verification as "building the product right" and validation as "building the right product" [29].

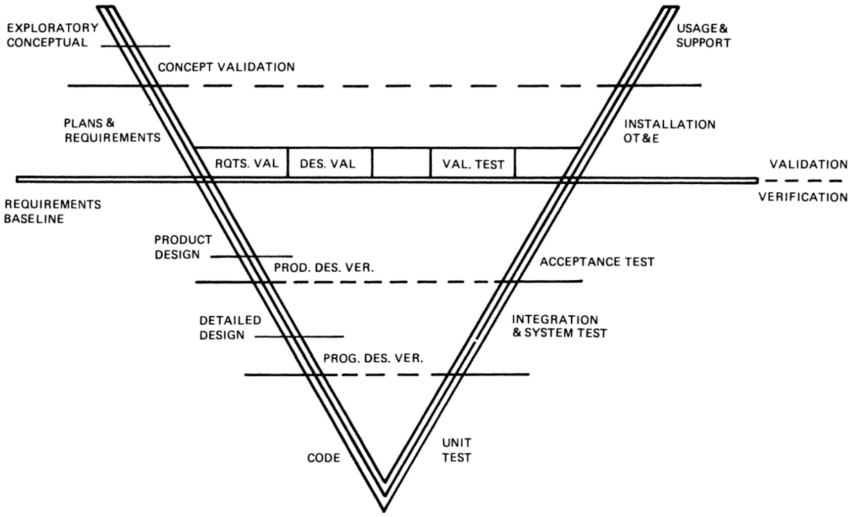


Figure 3.8: V-Model [27]

Unit, integration, system, and user acceptance tests

It can also be seen in Figure 3.8 that the V-model is strongly inspired by the Waterfall Model. Its stages are arranged in the left arm of the "V" from top-level specifications down to the implementation at the bottom. The key novelty is in the right arm: by segmentation of test levels in unit, integration, system, and user acceptance tests and juxtaposing these against every stage of the left side, a high level of test coverage is pursued since the outputs and artifacts of each stage on the left-hand side serve as a basis for the respective tests at the right side.

As a remark, Dolezel and Felderer have recently pointed out that this "concept of separate test levels with dedicated testing responsibilities codified by the V-model [...] has been traditionally presented as a form of test maturity ideal" since "the more test levels exist in the organization and the higher number of diverse groups involved in software testing, the more mature test process the organization exercises" [76].

Figure 3.8 also shows that verification and validation activities are separated by the "requirements baseline," which refers to the requirements specification elaborated and validated during the "Plans and Requirements" phase. This specification is developed in an iterative validation process, shown in the top half of Figure 3.9, and must be approved within a "Plans and Requirements Review" to serve as the basis for the software development contract between the customer and the software company.

On the other hand, verification is established by comparing the requirements baseline and all successive refinements elaborated in successive stages. Thus, verification activities begin in the "Product

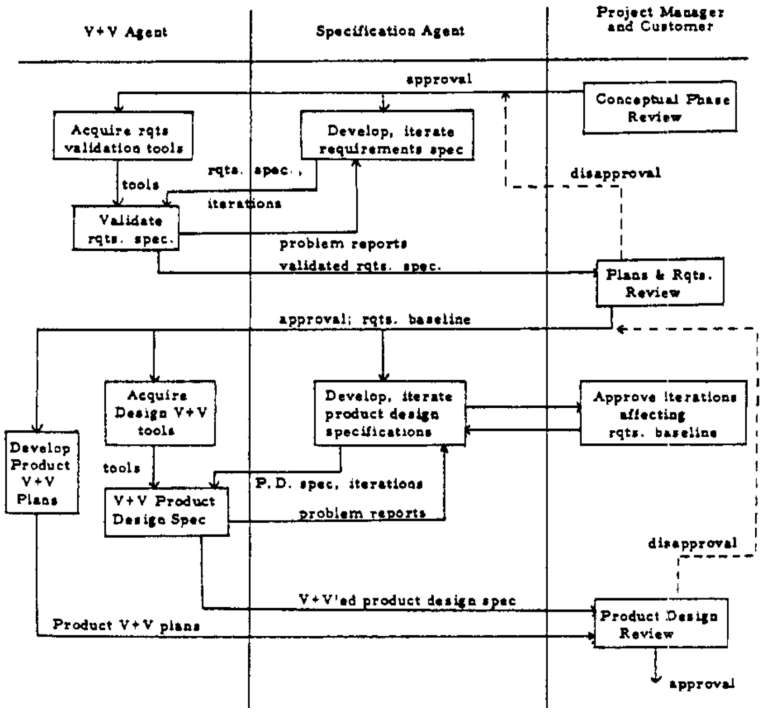


Figure 3.9: Roles of the V-Model [27]

Design" stage on the left side and conclude with the "Acceptance Test" on the model's right side.

In his paper, Boehm is very clear about the consequences and writes that verification activities "do not lead to changes in the requirements baseline; only to changes in the refinements descending from it," whereas "validation identifies problems which must be resolved by a change of the requirements specification."

That is also why he admits that the separation between validation activities above the requirements baseline and verification activities below the baseline is not as strict as the model suggests. Instead, there must also be validation activities throughout the whole lifecycle, including the development phase, which he clarifies by the following:

"For example, a simulation of the product design may establish not only that the design cannot meet the baseline performance requirements (verification), but also that the performance requirements are too stringent for any cost-effective product designs, and therefore need to be changed (validation)."
— Barry Boehm [29]

Besides this, the V-Model is very clear and precise about formal processes, as can be seen in Figure 3.9, which also shows that Boehm introduced specific *roles* ("V+V Agent" and "Specification Agent") to software projects. Even more, his paper included precious assistance for practical application, such as four elaborated criteria for require-

ments and design specifications (completeness, consistency, feasibility, and testability), a taxonomy of a satisfactory software specification, an evaluation of concrete verification and validation techniques, as well as recommendations for small, medium and large specifications including checklists.

In conclusion, the V-Model became widely adopted and still is the official development methodology of the German government (although in a slightly different form).

3.4.3 *Spiral Model*

The Spiral Model introduced iterations and refining prototypes

Seven years after introducing the V-Model, Barry Boehm published his paper "A Spiral Model of Software Development and Enhancement" in 1986 [30], which was republished in 1988 to make the *Spiral Model* known to a broader audience [31]. Boehm modified the Waterfall Model by introducing several *iterations*, which spiral from an initial software concept over several prototypes to the final development of the system (see Figure 3.10). By this sequence of refining prototypes, the Spiral Model emphasized an iterative development concept [213] (see Section 3.3), which, according to Boehm, represented a paradigm shift from a document-driven process to a risk-driven approach [30].

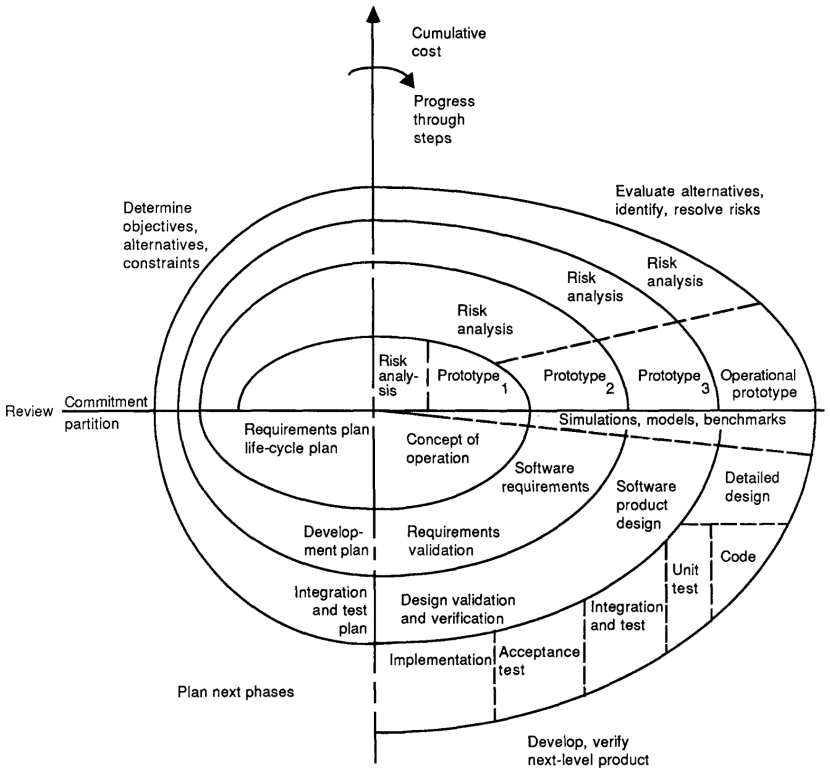


Figure 3.10: Spiral Model [31]

As can be seen in Figure 3.10, each spiral traverses four quadrants:

1. Determine objectives, alternatives, constraints
2. Evaluate alternatives, identify, resolve risks,
3. Develop, verify next-level product,
4. Plan next phases.

In addition, the radial dimension represents the cumulative cost incurred over the project term, while the angular dimension represents the progress made in completing each iteration cycle of the spiral. Boehm also points out that the model "reflects the underlying concept that each cycle involves a progression that addresses the same sequence of steps, for each portion of the product and for each of its levels of elaboration, from an overall concept of operation document down to the coding of each individual program" [31].

It is immediately apparent that Boehm's model builds up on the Waterfall Model, which is easily identifiable in quadrant three, where a prototype is designed, coded, verified against its requirements, and validated through testing. The innovation, however, is in quadrant two because prior to the building of the prototype, there is an upstream identification and analysis of risks, which are possible situations or events that may cause a project to fail and not meet its goals. The set of risks is manifold and ranges from trivial to fatal and orthogonal to that from improbable to very certain risks. Therefore, risk management includes a prioritization strategy depending on the impact and likelihood of problems and a mitigation strategy to deal with identified risks. The overall goal is to lower development costs by early elimination of nonviable alternatives to avoid unnecessary rework. Hence, risk management is used as a tool to compare the relationship of costs to the expected improvement and thus to determine the amount of time, resources, and overall effort to expend for all activities within the cycle [227].

In this context, it is also essential to mention quadrant four, especially the review that occurs at the completion of each cycle. Its objective is to investigate all artifacts produced during the previous cycle and compare them to the plans for the next cycle, including the allocation of resources. This review ensures that stakeholders are committed to the decisions and resulting activities of the succeeding cycle.

However, Boggs points out that although the Spiral model "does include the user representative as an input to the success of each cycle, it does not provide a strong vehicle for end-user involvement" [36]. He mentioned that Boehm addressed this issue to some part in 1994 by introducing his "Next Generation Process Model" (NGPM) [33], which "involved the stakeholders to a much larger degree and

better identified user's needs at the outset of the project." However, it still remained fairly possible that conflicts among stakeholders would occur. Therefore, Boehm released another iteration in 1998, twelve years after the Spiral Model's introduction, called the "Win-Win Spiral Model" [35]. This model added specific "Theory W"-activities to each spiral to identify the stakeholders and address their win conditions explicitly. Theory W is a management theory and approach that says making winners of the system's key stakeholders is a necessary and sufficient condition for project success.

While the Spiral Model became widely adopted in the industry, Boehm pointed out that it was often misunderstood. In a special report about a workshop on spiral development experience and implementation challenges held at the University of Southern California in 2000, he listed four main misconceptions that should be avoided [34]:

1. the spiral is just a sequence of waterfall increments,
2. everything on the project follows a single spiral sequence,
3. every element in the diagram needs to be visited in the order indicated,
4. there can be no backtracking to revisit previous decisions.

According to Boehm, these misunderstandings result from "oversimplifications" in the initially published Figure 3.10 and could fit only a few rare risk patterns but are not valid for most.

Instead, he clarified that risk management targets "lowering development cost by early elimination of nonviable alternatives and rework avoidance," which means that the result of planning and risk analysis should lead to different processes for different projects. Hence, "the spiral model is actually a risk-driven *process model generator*, in which different risk patterns can lead to choosing incremental, waterfall, evolutionary prototyping, or other subsets of the process elements in the spiral model diagram" [34].

3.5 ITERATIVE AND INCREMENTAL DEVELOPMENT

Section 3.3 already explained that linear and sequential SDLC models are nowadays described as "traditional," whereas the agile model is considered a new paradigm for software development. But how did this paradigm shift occur? And what is so special about it?

This section will show that this paradigm shift was not abrupt but a gradual process transformation with linear, sequential models on one end of the spectrum and agile methodologies on the other. Moreover, it

will show that the combination of *iterative and incremental development* (IID) has triggered the evolution.

First, this chapter presents today's understanding and key characteristics of IID before elaborating on the historical context and describing its transformative power to succeeding SDLC models and software development processes.

3.5.1 *Today's Understanding of IID*

Iterative and incremental development approaches have been briefly presented in Section 3.3 in the context of various SDLC models. Here, they are presented in direct comparison to understand that the transformative power of the paradigm shift towards agile methodologies (presented in the next section) was only possible by combining both.

Jeff Paton beautifully illustrated the differences between iterative and incremental development in his now famous talk "Embrace Uncertainty," presented at the "XP Day 2007" in London, which in a summarized form was later published as an article on his blog [205]. Inspired by John Armitage and his illustrations of the Mona Lisa in his paper "Are agile methods good for design" [10], Patton adapted this vivid example to explain the differences as follows.

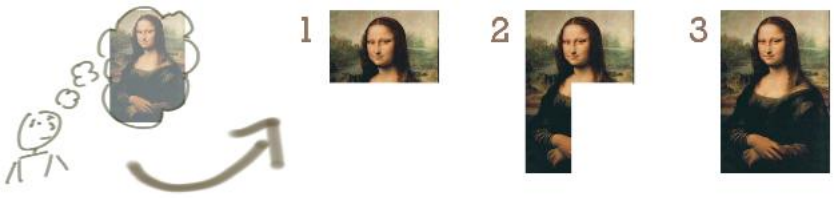


Figure 3.11: Incremental development [205]

Incremental development

According to Patton, *incremental* development creates software by a piecewise delivery of increments that add up to the final product - similar to building a wall by adding bricks. By this, two aspects are worth mentioning. At first, this means that every brick (the increment) is finished once added to the wall. It was shaped (developed) to fit a particular gap but will not be touched again once set. Like in the image above, notice that every piece that is added to the Mona Lisa is finished and complete, and, apart from adding glue to its borders (representing the integration into the product), will not be modified in terms of what has been drawn (what has been developed according to the user requirements).

Second, due to the completeness of each increment, there must be a detailed understanding of what is being built, and this must exist before any execution action. Speaking of the Mona Lisa example,

the artist (as the originator) would need to constantly have the final painting in his mind or (as a restorer or copyist) use a template, similar to a bricklayer who builds the wall according to a *predetermined plan*.

Especially this last aspect can also be identified in the definition of Alistair Cockburn, who described incremental development as a "staging and scheduling strategy in which various parts of the system are developed at different times or rates, and integrated as they are completed" [54]. Here, the emphasis is on "staging strategy," which explains the need for extensive upfront planning.

Figure 3.12: Iterative development [265]

Iterative development

On the other hand, *iterative* development is almost the exact opposite. As can be seen in Figure 3.12, each iteration of the Mona Lisa delivers a complete image, which is not split into parts and does not grow by separately developed increments, but rather the image as a whole is modified and refined with ongoing iterations. Again, there are two essential aspects to be aware of.

The first is that this approach assumes that *a product cannot be built in perfect condition and to total satisfaction by just a single attempt*, as it is almost impossible to get everything right according to the customer's needs on the first try. Instead, not just a few but several attempts are necessary, each one delivering new insights, which then lead to adaptations of the product. On the other hand, this also means that *a rough mental image of what should be built is sufficient* to start the practical action since it sharpens throughout the course of development.

So, Cockburn defines iterative development as a "rework scheduling strategy in which time is set aside to revise and improve parts of the system" [54]. Here, the emphasis is on "rework strategy," which incorporates the two aspects mentioned above.

However, one fundamental distinction of iterative development has not been mentioned yet, and it does not show up clearly in Cockburn's definitions or Patton's explanations. To "rework" and "revise," there must be a gain in knowledge. This, in turn, is only possible through constant verification and validation. Verification or "building the product right" is a matter of the software creators. It is up to the artist and his expert knowledge to choose his brushes and colors wisely to guarantee a certain quality so that, for instance, colors do not start to fade after the Mona Lisa has been delivered.

On the other hand, validation or "building the right product" is a matter of constant feedback from the customers. Hence, iterating is fundamentally different from incrementing because every iteration starts with a new or modified set of requirements. This set has to be derived from *valuable*[3] external feedback; otherwise, an iteration would not make sense.

As separate approaches, incremental and iterative development have significant ups and downsides, but the combination of both becomes more than the sum of its parts. Steven Thomas has built up on the Mona Lisa example of Jeff Patton in 2012 and added an illustration of the combined approach, which is shown in Figure 3.13 [265].

Figure 3.13: Combined incremental and iterative development [265]

Combining iterative and incremental development

As can be seen, at every one of the six steps, the increment adds new features and expands the scope of the offered functionalities - this is the incremental part. But each increment also refines parts of the existing functionality - that, on the other hand, is the iterative part. In other words, due to the iterative aspect, the increment no longer needs to be complete and developed to its full extent. Instead, it can be rather abstract since future iterations will sharpen it. Furthermore, to turn it around, due to the incremental aspect, the refinement of an iteration must no longer consider the whole product but only the parts of concern.

Now, back to the claim that this IID is more than the sum of its parts and hence can be seen as the source of the evolution from sequential SDLC models towards modern agile development processes. Sticking to the Mona Lisa example, the overall development process might start with a specification from a customer saying, "I want you to paint a beautiful lady with a mysterious smile in a landscape setting." The benefits of IID and their meaning for a software project can be seen right at the first step of Figure 3.13.

3 The word "valuable" is emphasized to highlight and show the connection to the four agile values (see Chapter 3.6.4), one of which is "customer feedback."

Even with this rough specification, the development process can start. In contrast to pure incremental development, there is no need for a complete upfront specification so that a first version of what has been understood so far can be created quickly. Notice that it is yet unknown which parts of the body will be important to the customer. However, since the smile was explicitly mentioned, the face was considered a primary concern, and, therefore, more time was spent on its details.

In case of feedback from the customer that the outcome is entirely wrong or if the customer loses interest in the project, money is lost for the work to come that far. However, the first outcome could be validated *early*, and only a few resources have been spent compared to the big upfront specification of pure incremental development (see Step 0 of Figure 3.11), which is in no way guaranteed to be free of misunderstandings or errors. On the other hand, in comparison to pure iterative development, the customer can give feedback specifically to the main important parts. For instance, in Step 1 of Figure 3.13, a first draft of the color concept can be evaluated, whereas Step 1 of Figure 3.12 does not show colors at all.

In addition, the flexibility and new possibilities of IID can also be seen in Step 2 of Figure 3.13. As a result of the validation of the previous step, the customer may think about whether just the face of the lady could be enough, so he may decide to focus on the face to a greater extent and postpone the decision concerning the less important parts to a later point in time. At least he already got an impression of these parts and how the overall painting could look like. Step 2 of Figure 3.13 now shows the entirely new possibilities of IID in the form of new choices for the customer. If he decides that the top priority functionality (face of the lady) is enough and satisfies his wishes, the project can stop with a good return on investment. It is not perfect since time and effort were spent on less critical parts of the image, which became redundant by his decision to stop the project. However, the extra amount is acceptable and much lower than it would be by either one of the individual development approaches.

In summary, IID is characterized by much better *risk control* since there is no need to understand and address the complete scope at the very beginning of a project. A rough idea is sufficient so that the building process can start earlier. Short iterations then promote constant feedback loops, following a strategy that cancels out the disadvantages of the individual approaches, thus incorporating the benefits into a new, very flexible, and risk-on-investment-balanced development process.

3.5.2 *IID as Transformative Power to the Paradigm Shift*

Incremental and iterative development (IID) is a fundamental aspect of the agile SDLC model and some of its methodologies, which will be presented in the following section. In fact, by investigating the history and evolution of software development processes, IID reveals itself as a condition sine qua non for the fulfillment of inherent properties of software projects that have led to the software crisis (see Section 2.2.2). Therefore, to understand its importance, it is worth looking at the influence of IID on the progress of software development processes and how it acted as a transformative power to evolve from sequential, document-driven processes towards today's agile methodologies.

Larman and Basili have shown that the roots of IID can be traced back decades ago and that the principles of iterative and incremental development were the "common theme" underlying various upcoming SDLC models - all of them very different but with the mutual goal of avoiding a sequential one-way and document-driven development process with gated steps [155].

They dated IID as far back as the *Project Mercury*, which was the first human spaceflight program of NASA in the early 1960s, and quoted Gerald M. Weinberg, who worked on the project:

"All of us, as far as I can remember, thought waterfalling of a huge project was rather stupid, or at least ignorant of the realities... I think what the waterfall description did for us was make us realize that we were doing something else, something unnamed except for 'software development.'" [155]

According to Larman and Basili, this "unnamed" practice was, in fact, an IID approach, making use of very short iteration cycles of just half a day and incorporating aspects of what later became *Extreme Programming* (see Section 3.6.3).

In order to follow the historical evolution of IID as the transformative power of SDLC models, Section 3.4.1 has already explained that Royce was the first to openly address problems of the Waterfall Model in 1970 and proposed a "do it twice" approach, by adding an iteration to the otherwise strictly sequential development process. Although this by itself is not IID, his solutions also show the first signs of a feedback-based iterative step, representing a very early form of adaptation [155].

Speaking of the 1970s, Larman and Basili also identified several approaches to IID within life-critical military projects of the U.S. Department of Defense. However, all of them still had significant upfront specification effort. The first example describes the development of

a command and control system for the first U.S. Trident submarine (a project with more than one million lines of code), which had to be delivered by a specific date, or the executing company would face a penalty of $ 100.000 for each day the deadline is exceeded. The team addressed problems of managing the risks and the complexity of this large-scale development by dividing the process into four time-boxed iterations of six months each.

A second example describes an extensive application of IID during the development of the *Light Airborne Multipurpose System* for anti-submarine warfare helicopters of the U.S. Navy, a project (of several millions of lines of code) that was successfully and incrementally delivered within 45 iterations and for the first time made use of an iteration length of one month, which is in the proposed range of later appearing agile development methodologies. Furthermore, Larman and Basili mention a project for ballistic missile defense conducted by TRW (a company in the defense industry and where Royce and Boehm both worked simultaneously), which made use of feedback-based iterations, as well as NASA's space shuttle and its primary avionics software system as a "striking example of a major IID success" exhibiting several of the IID characteristics, such as short and time-boxed iterations, small incremental releases and feedback-driven refinement of requirements [155].

The concept of IID began to spread during the 1970s

These examples show that aspects of IID already have been applied before and during the 1970s, although primarily within governmental projects "behind closed doors." However, due to the software crisis (as explained in Section 2.2.2) and the inflexibility of the existing Waterfall Model to deal with prevalent problems, researchers and practitioners began to publish novel ideas for better software development and spoke to a broader audience, so that the idea of IID began to spread.

This can be shown by several publications, especially during the 1980s, when the Waterfall Model was massively called into question.

The shift from "rework" to "evolutionary delivery"

In 1976, *Tom Gilb*, one of the most active IID promotors, introduced his practice of "evolutionary project management" and initiated that not only the concepts of software development processes have changed over time, but the meaning of the phrase "iterative development" has been subject to change as well. He invoked that it evolved from basic "rework" (see the "do it twice" approach of Royce on Page 39) to the understanding of modern IID and its implication of "evolutionary advancement" [155], which can be seen in the agile development context including light and adaptive iterations, where iterating is not just revisiting work, but rather also the possibility for constant innovations. His massive questioning of sequential software development can best be seen by a statement from his paper "Evolutionary Delivery versus the Waterfall Model" [95], in which he said that "the 'waterfall model' may be unrealistic, and dangerous to the primary objectives of any

software project." Instead, it needs "evolutionary delivery" with iterative and incremental software development, delivered to the customer every few weeks to obtain feedback. His final ideas were later published in 1988 in his book "Principles of Software Engineering," which, according to Larman and Basili, is "the first book with substantial chapters dedicated to IID discussion and promotion" [155].

Further reconsiderations of the Waterfall Model can, for instance, be found in the paper "Life-Cycle Concept Considered Harmful" by McCracken and Jackson from 1982 [175], where the interesting title is an homage to Dijkstra's classic "GoTo Statement Considered Harmful" [75] and furthermore shows that during this period the Waterfall Model was so dominant in use that it was synonym to "lifecycle."

Widespread criticism of sequential models and the promotion of IID during the 1980s

Besides this, two landmark publications promoted IID during the 1980s. The first was Boehm's Spiral Model [30], already presented in Section 3.4.3 as the first formalized approach using several iterations based on risk assessment. The second is Frederick Brooks's "No Silver Bullet" paper [38], in which he commented on the predominant sequential way to develop software by saying:

"Much of present-day software acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software acquisition problems spring from that fallacy." [38]

Afterward, he treats iterative and incremental development as "promising attacks on the conceptual essence" and states:

"Nothing in the past decade has so radically changed my own practice or its effectiveness." [38]

According to Larman and Basili, Brooks's rejection of sequential development processes intensified over several years and finally culminated in a clear opinion, which shows in his keynote speech at the "1995 International Conference on Software Engineering" entitled *"The waterfall model is wrong!"* [155].

These examples illustrate that the 1980s were a decade of massive questioning of sequential development processes. Nevertheless, projects starting to incorporate aspects of IID still struggled to get rid of extensive preliminary upfront specifications [155]. This controversy can also be seen in the publication "A Rational Design Process: How and Why to Fake It" by Parnas and Clements in 1986, in which the authors list many reasons why thorough specifications before development are "impractical" and "unrealistic," but nonetheless necessary and part of a "rational, ideal software design process" [204].

Arising IID methodologies during the 1990s, which later will be coined "agile"

During the 1990s, the renunciation of what are now called "traditional" SDLC models accelerated, and the rather theoretical concept of IID was incorporated into practical application in the form of new arising process methodologies. Many of these should later be called "agile," such as *Rapid Application Development* (RAD, see Section 3.6.1, *Dynamic Systems Development Method* (DSDM, see Section 3.6.2), *Extreme Programming* (XP, see Section 3.6.3), *Feature-Driven-Development* (FDD) or *Scrum* (see Chapter 4).

However, not all approaches incorporating the ideas of IID have later become "agile." To be very clear, the concepts of IID are necessary for an agile process, but as will be shown in the following section, there is more to it than just the combination of incremental and iterative development.

As an example, the *Rational Unified Process (RUP)* introduced by Philippe Kruchten [147] is one of the upcoming IID approaches of the 1990s that is not considered to be agile. Although the combination of iterative and incremental development is one of its main components, RUP is very formal and describes a phase-driven software development process that still needs a certain amount of upfront specifications and analysis. In addition, it is use-case-driven and model-heavy because it is closely linked to the invention of the *Unified Modeling Language* (UML), which was simultaneously developed by representatives of object-oriented programming known as the "three amigos" Grady Booch, Ivar Jacobson, and James Rumbaugh, to unify various existing software specification notations [142].

To conclude, while the 1980s were the decade of massive questioning of the traditional, sequential, and document-driven software development process, the 1990s were the decade of transforming that questioning into new methodologies, all of them with individual process definitions but mutually sharing the fundamental combination of incremental and iterative or "evolutionary" development.

In 1994, the often cited "CHAOS Report" from the Standish Group [105] finally underpinned the substantial significance of IID to software development. By investigating failure factors of more than 8.000 projects, the report identified sequential development practices and complete upfront requirements specification as top reasons for project failures in scope and budget. In the end, a key conclusion proposing the adoption of IID can be found in the report's penultimate paragraph:

"Research [. . .] indicates that smaller time frames, with delivery of software components early and often, will increase the success rate. Shorter time frames result in an iterative process of design, prototype, develop, test, and deploy small elements. This process is known as 'growing' software, as opposed to the old concept of 'developing' software.

Growing software engages the user earlier [...] and expectations are realistically set. [...] Making the projects simpler is a worthwhile endeavor because complexity causes only confusion and increased cost." [105]

After this conclusion, the report closes with the final statement:

"There is one final aspect to be considered in any degree of project failure. All success is rooted in either luck or failure. If you begin with luck, you learn nothing but arrogance. However, if you begin with failure and learn to evaluate it, you also learn to succeed. Failure begets knowledge. Out of knowledge, you gain wisdom, and it is with wisdom that you can become truly successful." [105]

This quote is of particular interest since it anticipates a central aspect of the rising agile mindset, which further extends Gilb's concept of IID as a form of "evolutionary advancement" - and that is giving people the freedom to experiment and to make mistakes.

The next chapter will elaborate on the agile mindset, the incorporated values and principles as a distinction to IID, and some of the new methodologies mentioned before.

3.6 THE BIRTH OF "AGILE" AS THE CURRENT STATE OF THE ART

Driven by the growing awareness about the necessity to transform the existing way of software development into something more flexible, resistant to risk, and adaptable to different kinds of projects, many so-called "lightweight" approaches were published during the 1990s when the theoretical concept of IID was incorporated into different forms of practical application.

Some of them are briefly presented in the following sections to give an impression of their different focal points and areas of emphasis. At the same time, all share a common philosophy that will be coined as "agile values" some years later.

3.6.1 *Rapid Application Development*

The approach *Rapid Application Development* (RAD) was published in 1991 within the same-titled book by *James Martin* [168]. He developed it during his employment at IBM in the 1980s as a modified version of the existing methodology of Scott Shultz called *Rapid Iterative Production Prototyping* (RIPP), which in turn was heavily influenced by previous work of Barry Boehm and his Spiral Model (introducing software prototyping as a way of reducing risk, see Section 3.4.3) and Tom

Gilb's concept of evolutionary development (where a prototype is grown and refined iteratively into the final product, see Page 3.5.2).

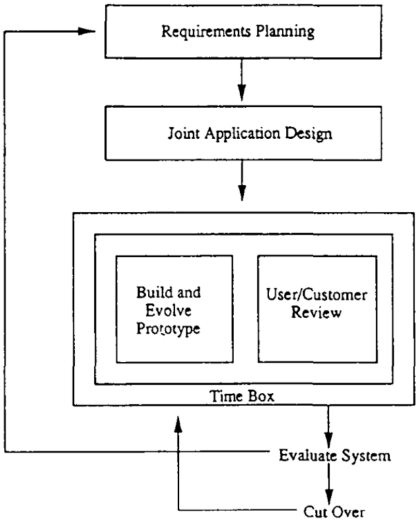


Figure 3.14: Rapid Iterative Production Prototyping (RIPP) [295]

RIPP promoted JAD sessions as the first operationalization of user involvement

As shown in Figure 3.14, the RIPP development process is defined as follows. It starts with the usual requirements planning but is immediately followed by a novel *Joined Application Design* (JAD) session. JAD itself also originated at IBM in the late 1970s to target the problem of insufficient user involvement in existing software development processes [49]. Although, at that time, there was general agreement that involving users in the entire software development lifecycle would lead to more successful systems, there were still no models nor operationalization of user involvement in the actual development lifecycle. In practice, users have, therefore, been only included for a matter of approval at discrete points in the development process, for instance, in signoff meetings, but not as active members of the creative process.

For that reason, JAD was created to tackle existing problems, such as the great challenge of compressing the development time while simultaneously handling fluctuating requirements and satisfying the need for innovative system design. As a solution, JAD introduced facilitated meeting structures to the development lifecycle, necessitating early and rigorous user involvement. These meetings aim to gather all relevant people and decision-makers in one place simultaneously so that participants can share their individual viewpoints and expertise. With a focus on encouraging creativity through collaborative brainstorming and because of arising synergies and new dynamics of this group work setting, users have been the driving force for specifying the requirements and design details, leading to faster development and innovative system designs [49].

Besides the analysis and system design, RIPP also includes users in the actual development phase, which follows an iterative and incremental approach and makes use of development cycles that are restricted to a certain amount of time (time-boxed), during which software is built through the development of prototypes, which become production subsystems if user/customer consensus is obtained [295]. That makes user involvement a critical method for prototyping and managing the risks of potentially changing requirements.

Meant to be a progression of RIPP, James Martin based *Rapid Application Development* (RAD) upon this development process, including the JAD session and iterative prototyping, but adapted it "to take maximum advantage of powerful development software that has evolved recently" [168].

In comparison, both approaches focus on a much faster development time and results with higher quality than those achieved with the traditional lifecycle model. However, RIPP only emphasizes building working prototypes that can handle data input so that users can evaluate what will become the final product. In contrast, RAD is more stringent *how* prototypes must be developed. For instance, development teams must consist of 2-4 so-called *SWAT*[4] developers - meaning people who are highly capable of using the most advanced CASE[5] tools of that time.

RAD stipulates the use of CASE tools

These tools were meant to change the method of software development by resolving the dependency between code quality and individual developer skills through design-automation techniques, automatic generation of executable code, and computer-aided planning and analysis, like visual data and process modeling. Thus, as a progression of RIPP, RAD is more specific on its methodology, the people involved, the overall management, and the tools being used and, therefore, claims to be much faster and less error-prone so that the development of an arbitrary system would not take longer than 120 days [122].

Since its introduction in 1991 by Martin's methodology, the term "Rapid Application Development" and its acronym RAD have been used in the community in a broader, more generic sense, describing various techniques and methods for speeding application development through prototyping and software frameworks. However, the problem with this was that the defined process of Martin's RAD was soon misunderstood as a "quick and dirty" approach without the need for technical cleanliness, as stated by Paul Herzlich[6]:

4 Specialists With Advanced Tools
5 Computer Aided Software Engineering
6 Inventor of the W-Model as a successor of the V-Model (see Section 3.4.2) and advocate of testing all artifacts in a staged process - both documentary and software.

"RAD's flashy name leads to misunderstanding. It is easily perceived by a new generation of developers as development with racing stripes. It sounds enticingly free from the shackles of the traditional methodologies - documentation, reviews, test plans, signoffs - all the things which programmers generally hate, and quality assurance depends on." [115]

In addition, Herzlich says that the confusion about RAD was also reinforced by vendors of modern CASE tools for rapid prototyping by massively using the sonorous terms "RAD" and "Rapid Application Development" in their marketing campaigns, hence portraying RAD as simply a matter of choosing proper development tools.

"It may be true that all these tools climbing on the RAD bandwagon facilitate prototyping and allow you to write programs faster, but their capabilities are not, by a wide margin, sufficient in total for delivering fit-for-purpose systems faster, which is the real akin of RAD." [115]

False marketing led to RAD being misunderstood as a "quick and dirty" approach

In reality, buying and using specific CASE tools also meant becoming dependent on the development processes prescribed by the tool vendors, which led to myriads of different implementations of what was marketed as RAD. So, in conclusion, the contradicting situation with great marketing hype about RAD by the tool vendors on the one side but no defined standard for an iterative process supporting the new kind of development on the other, the acronym became negatively connoted, and many jokes about what RAD stands for were made, like "rapidly achieving disaster" or "really awful design" [256].

Nevertheless, although the term RAD was misunderstood and used as a general alternative to the Waterfall Model, the approach by James Martin laid a cornerstone for widespread interest in the core of RAD. Rapid prototyping and including the user in the development process, together with prototypes that incrementally evolve into the final product, enabled faster feedback-response cycles and mitigation of risks, resulting in faster development projects with a much higher chance of staying within budget.

3.6.2 *Dynamic Systems Development Method*

Against the backdrop and circumstances that RAD was heavily misunderstood as a "quick and dirty approach," a group of sixteen RAD practitioners met in the UK in January 1994 in order to discuss a new standard following some of the original principles of Martin's approach [155]. This group consisted of information systems professionals from small and large organizations and project managers from some of the biggest IT companies, such as IBM and Oracle [256]. Together, they formed a not-for-profit Consortium, which dedicated its

DSDM was envisioned to better promote the misunderstood concepts of RAD

work to "develop and evolve continuously a public-domain method for rapid application development" [256], which should be specified in a way that can be widely taught and promoted through provided training and certification courses [179]. The outcome was the *Dynamic Systems Development Method* (DSDM), which provided a framework of controls and best practices for developing high-quality business applications.

Due to its high influence on other agile methodologies, DSDM will be captured here in more detail.

The following figure shows the first version of the DSDM development life cycle, also known as "three pizzas and a cheese," which was released in 1995 and quickly gained popularity. Over the years and by following feedback from adopters of the framework, the DSDM consortium also addressed particular needs and, therefore, made some additions by publishing White Papers, for instance, about how to incorporate the use of UML in DSDM projects [62]. However, the core principles, which are described in the following, remained the same.

The lifecycle is divided into five main phases, as illustrated by the darker shadings in Figure 3.15, framed by a prior pre- and succeeding post-project phase (not shown in the figure).

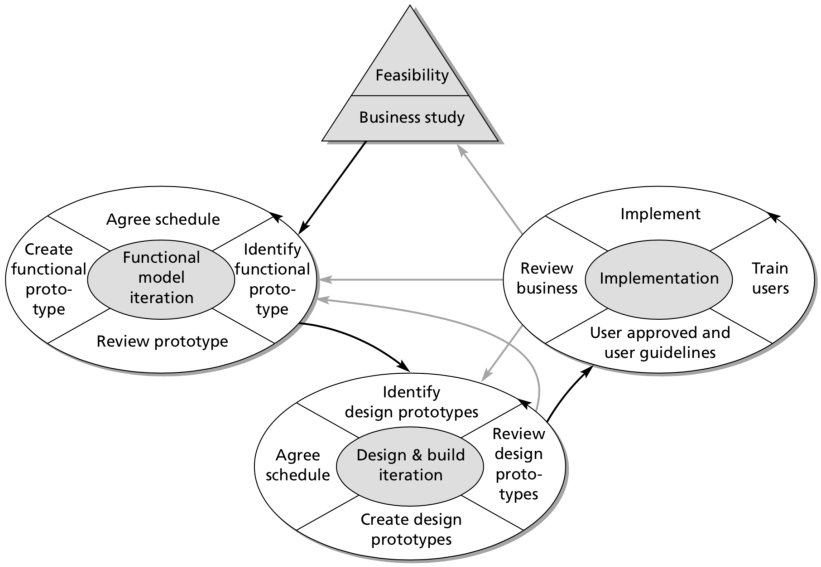


Figure 3.15: DSDM [256]

A feasibility study identifies potential project risks

Once the *pre-project phase* has led to a positive decision about a project, e.g., by ensuring that funding is available, a *feasibility study* is conducted in order to identify potential risks to DSDM success that need to be addressed upfront and to determine whether or not a prospective project complies with the criteria for DSDM and its RAD principles. The goal is to take a realistic look at the starting position,

for instance, by investigating the working environment and the people involved to decide whether the project is likely to be feasible from a technical perspective and appears to be cost-effective from a business perspective. This upfront effort lasting no longer than a few weeks should be just enough to decide on tailoring DSDM for the given situation and to set the project up correctly or to stop it right at the beginning if it turns out to be not viable [258].

A business study investigates use cases and outlines a draft for later development

After this, provided that the project qualifies, a *business study* is conducted, in which the business area is analyzed deep enough within highly collaborative facilitated workshops between knowledgable staff and decision-makers. The outcome is a *Business Area Definition*, representing a high-level view of the process, for instance, in the form of data-flow and entity-relationship diagrams or business object models with outlined use cases, which all together outline a draft for later development, including a list of *prioritized features* to be implemented. Besides understanding the functionality from the business perspective, this phase is also concerned with decisions about fundamental technical requirements in preparation for the succeeding development iterations, such as the *System Architecture Definition* describing the development and target platforms, as well as a *Development Plan* covering strategies about prototyping and testing activities.

DSDM differentiates between two development phases: the *functional model iteration* and the *design and build iteration*. As can be seen, both define a feedback-response cycle consisting of four activities:

1. Identify requirements and define what should be done,
2. Agree on a development strategy and schedule,
3. Create or refine prototype,
4. Check result against requirements and review outcome.

What the image of the lifecycle is not showing precisely, but what should be noted is that both iterations imply *several* of these feedback-response cycles per iteration, i.e., the activities above are repeated as long as the review does not release a satisfactory solution, which serves as input for the next phase.

The functional model iteration is about prototyping and defining primary functionality

Regarding content, the functional model iteration is about *analysis models* and *software components* containing the primary functionality. These are built side by side and mutually influence each other so that the findings of prototyping activities feed back into extended analysis models, which in turn trigger the refinement and testing of prototypes so that they progressively move towards potentially releasably software. However, these prototypes might not be engineered to the deepest extent to be truly releasable; for instance, performance optimizations could have been left out.

The design and build iteration evolves the functional prototypes into truly releasable software components

For that reason, the content of the *design and build iteration* is about bringing the system to a sufficiently high engineering standard in order to start the implementation process (notice that in DSDM, "implementation" refers neither to "coding" or "development" activities, but rather to deploy the system and prepare it for use). Mainly, this includes testing non-functional aspects and triggering cycles that let the functional prototypes evolve into truly releasable components as input for the implementation phase. Notice that in the lifecycle diagram, there is also a backward arrow from the review activity to the identification of requirements in the functional model iteration. That is because it may be more sensible to address particular functionalities together with related designs of non-functional aspects, depending on the tools and technologies of the development environment and how the application breaks down into separate components. In that case, functional and design prototypes may evolve simultaneously.

The implementation phase puts the development into operational use

Finally, the *implementation phase* serves as a cutover from the development environment to the operational environment, which includes approval by accountable senior users of the team and the creation of necessary documentation for being able to train end-users as well as persons of the system's future operational support once it is deployed and running. While this phase is usually conducted once, it may also iterate in case of a dispersed user population, for instance, in case of major system rollouts across multiple nations [62]. Once the system is rolled out, the lifecycle is concluded by a review activity resulting in an *Increment Review Document*, which summarizes the achievements of the project and compares requirements that have been identified during development against the implemented software components. Depending on the review, subsequent improvements may be identified, re-triggering parts of the DSDM lifecycle, as illustrated by the back arrows of the implementation phase in Figure 3.15.

For instance, during development, it could be discovered that an important area of business functionality was not considered, which was then deferred to meet the delivery date. In that case, returning to the business study is necessary to scope another development cycle. More probable, however, is that either lower prioritized functionality or non-functional aspects have been omitted during development because of time constraints so that the delivered system still satisfies in the short term but should be refined for long-term use by returning to the respective functional model or design phases.

The nine DSDM principles

The framework is based upon the following *nine principles*, and the DSDM specification is very clear about their importance, saying that "if one of them is ignored, the whole basis of DSDM is endangered." Moreover, even if some projects may find it hard to apply all of these principles and would like to drop only a few, the inventors

make clear that in those cases, "the use of DSDM should be seriously reconsidered" [256, p. 11].

While the first four principles define the very core foundations and the philosophy on which DSDM is built, the other five guide the structure of the framework's development process.

1. **Active user involvement is imperative.**

   The inventors stress that user involvement is not only active but "pro-active" [62] in the sense that throughout the project, a few senior users support and actively participate in a development team to ensure that development is continuously heading in the right direction by providing detailed knowledge and in-depth understanding about the underlying business and its processes.

2. **DSDM teams must be empowered to make decisions.**

   With expert users in the team knowing what business requires and continuously providing feedback, as written above, team members must be empowered to make their own decisions about the direction to take for development. If decisions would need to wait for approval and move up and down the management hierarchy, the involvement of expert users to speed up development would not make sense. As a result of this new empowerment, managers have to deal with losing control, a major issue of agile approaches in general, as will be shown in later chapters. However, DSDM provides a structure for "escalating decisions" and distinguishes between smaller and frequent decisions that can and should be made by the team and more fundamental decisions with severe consequences that are within the responsibility of senior management [62].

3. **The focus is on frequent delivery of products.**

   By this principle, DSDM emphasizes a product-based approach, but what is meant by "products" is not completely operational systems. Instead, it is about *components* of the final software (or increments in terms of IID, as explained in Section 3.5), in the sense of something tangible (e.g., the data model of the application) that can be verified as acceptable by staff outside the team. By frequently delivering and verifying a new component or part of the business application, management regains control of the project's direction and can adjust if the development team makes erroneous decisions.

4. **Fitness for business purposes is the essential criterion for acceptance of deliverables.**

   With this formulation, the fourth principle of DSDM clarifies that developers must "build the right product before building the

product right." In other words, validation is more important than verification. So, instead of delivering "gold-plated" solutions, the focus should be on maximizing the business benefit. This means that minor (technical) issues are acceptable if they are less relevant and will not directly affect the business purpose.

5. **Iterative and incremental development is necessary to converge on an accurate business solution.**

   By this principle, the DSDM specification stresses a mindset assuming that rework is an integral part of the development process and must be accepted as something positive - a profitable result of continuously processing the business case and understanding it better. Furthermore, since actual users are part of the development team, it is possible to gather instant feedback and let systems evolve by iterative and incremental development so that errors are trapped very early, instead of taking a "one-shot" approach, where errors are not only costly to correct, but may endanger the whole system development.

6. **All changes during development are reversible.**

   Closely linked to the previous one, this principle is about proper process management to guarantee that the possibility of returning to a safe point of development exists so that wrong paths do not lead to vast amounts of discarded work. With this, it builds onto the third principle, the frequent delivery of verified components, and adds strategies in case of wrong development decisions.

7. **Requirements are baselined at a high level.**

   This principle means that the requirements identified during the business study serve as an agreement defining the high-level scope of the project. As a result, further details are determined during the following iterative prototyping activities. This prevents over-specification and guarantees that prototyping can start immediately, focusing on the essential aspects.

8. **Testing is integrated throughout the lifecycle.**

   With this principle, DSDM emphasizes that testing is such an integral part that it should never be done after development in a separate phase but rather in parallel to the actual coding process in the form of a "test as you go" philosophy. With developers testing software components for technical aspects and the users in the team testing for functional suitability, all forms of acceptance tests are carried out incrementally. In addition, DSDM stipulates that due to the evolutionary development of software components, integration and regression tests must be performed as soon as new versions evolve.

9. **A collaborative and cooperative approach between all stakeholders is essential.**

   In DSDM, responsibilities are shared, meaning that "collaboration and co-operation are important, but also that all stakeholders need to buy into the approach" [62]. For this purpose, not only must there be a close collaboration between developers and end-users to determine what is needed, but also cooperation between different parts of the company, such as business and IT organizations, in order to avoid barriers that arise when departments consider themselves as being independent of all the other. This principle also means that stakeholders must understand that the contract about what should be delivered cannot be cast in stone due to the iterative and incremental approach. For instance, when development works against time constraints and new important requirements arise that have not been considered before, they cannot be simply added to the workload. Instead, a compromise is needed, and stakeholders need to cooperate and agree on a modified list of what is essential to be delivered.

The MoSCoW prioritization of requirements

Besides these principles, another aspect of DSDM should be mentioned since it became popular for other time-constrained development approaches, especially for Scrum, which is the primary concern of this thesis. It is known as the *MoSCoW* prioritization of requirements, yet the "o's" have no meaning and represent filling letters around the acronym "MSCW," which stands for "Must have," "Should have," "Could have," and "Want to have but will not have this time round" [62].

Must have requirements are fundamental to the system and altogether define the minimal usable subset without which the system would not be able to work appropriately and, therefore, would be useless for the user. For example, for an online shop, customers must be able to complete the whole ordering process, including selecting items and making the payment via credit card.

Should have requirements are not mandatory since the system will be useful and usable without them. However, they add tremendous value to the user experience, so they are classified as the second most important features that should be implemented during the project. For instance, offering additional payment options, such as paying via invoice, bank transfer, or PayPal, would certainly add value to the user experience, but the ordering process would still be usable without them.

In contrast, *could have* requirements are "nice to have" but may be easily left out of the project. Usually, they will only be considered if the higher prioritized features can be implemented so flawlessly that time is still left over. Regarding the online shop example, the

possibility of seeing items in a 3D perspective before buying would be a big difference for the user. However, it is not an essential feature for the ordering process in general.

Lastly, requirements prioritized as *want to have but will not have this time round* represent features with a certain amount of value (for the stakeholders and users) but are not considered yet since they can wait for later development. For instance, the online shop could be adapted to different languages. It is not yet necessary, but it is certainly useful in the later stages of development.

While all requirements determine a full system, the MoSCoW rules provide a basis for decision-making about what will be covered during which time-box of a given project.

3.6.3 *Extreme Programming*

Besides Scrum as the most prominent agile development framework and the primary concern of this thesis, *Extreme Programming* (XP) has also become one of the most famous lightweight approaches in software development.

XP was created by *Kent Beck* in 1996 when Chrysler asked him to become the Project Lead on the Chrysler Comprehensive Compensation System (C3) [113]. The C3 project was born out of necessity because Chrysler's payroll accounting was handled by three different systems, each over twenty years old, requiring separate programming staff for maintenance and a separate customer staff to use them.

Previous attempts utilizing the Waterfall Model to develop a replacement for these legacy systems failed, so Kent Beck was hired to finally rescue the project and develop a new payroll system that should "allow Payroll Services [...] to more easily manage the requirements for accurate and timely service [Chrysler's] [...] 86,000 employees by reducing the duplication of effort the legacy systems require" [6].

XP proposes five team values and twelve development practices

Together with *Ward Cunningham* and *Ron Jeffries*, Beck established a new development process by taking *twelve best practices of software development* to the "extreme" and structuring them around *five values* [18], which later would strongly influence the basis of all agile methodologies, known as the "Agile Manifesto" (see Section 3.6.4).

The five values of XP are:

1. **Communication**: XP stresses the importance of clear and frequent communication among all stakeholders, including team members, developers, and customers. This ensures everyone works jointly at every project stage and is aligned on project goals, requirements, and progress.

2. **Simplicity**: Developers focus on creating the simplest possible solution that works and meets customer needs. This means avoiding unnecessary complexity and being willing to conduct code refactoring frequently to minimize risk.
3. **Feedback**: XP emphasizes that team members deliver software frequently to get feedback as early and often as possible, which helps to address issues promptly and align the project with customer requirements.
4. **Courage**: Team members are encouraged to take on challenging tasks, make necessary changes, and stand by their commitments, even under difficult circumstances.
5. **Respect**: Mutual respect among team members and stakeholders is crucial. Everyone's contributions are valued, and team members work in a cooperative and supportive environment to strive for a common goal.

Based on these values, XP proposes *12 software development practices*:

1. **The Planning Game**: XP proposes a two-step planning process addressing the crucial software development questions: what will be achieved by the deadline and what to do next.

   Release Planning involves the customer presenting desired features to the programmers, who then estimate the difficulty of implementing these features. Using these estimates and understanding the importance of each feature, the customer develops a project plan. Initially, this plan is imprecise as priorities and estimates may not be fully defined, and the team's pace is still unknown. However, the first release plan is sufficiently accurate for making initial decisions, and XP teams regularly update it every few months (see Figure 3.16 on Page 67) as the project progresses.

   Iteration Planning, on the other hand, is a bi-weekly meeting where the team receives direction from the customer for the next iteration. Programmers detail the proposed iteration plan into tasks and estimate their cost more precisely than in Release Planning before finally committing to the iteration's scope.

2. **Small Releases**: Software is developed in small, frequent releases to allow rapid feedback and course correction. First, this includes releasing running and tested software to the customer and delivering the proposed business value at the end of every iteration. Second, this also includes releasing to end users as often as daily to conduct acceptance tests (see Figure 3.16 on Page 67), receive feedback, and monitor how a feature works in production.

3. **Metaphor**: XP teams develop a shared vision of how the product works, called the "metaphor." For example, as explained by Ron Jeffries, an agent-based information retrieval system could be envisioned to work "like a hive of bees, going out for pollen and bringing it back to the hive" [221]. While such a poetic metaphor may not arise in any case, a metaphor should still use analogies to describe the system and its functionality to aid understanding and communication.

4. **Simple Design**: Design is kept as simple as possible, exactly fulfilling the system's current functionality but not over-engineering future possibilities. This means that design in XP is not a one-time activity but instead an ongoing process throughout the entire course of development.

5. **Test-Driven Development**: XP strongly emphasizes feedback, which highly depends on effective testing. To ensure that the developed software is always in a functioning state, teams utilize *test-driven development* that entails adding a test before working on even the smallest feature and making the test pass. This approach typically results in code with nearly 100% of test coverage. However, simply writing tests is not enough because they must be consistently run, which in XP is taken to the "extreme" because every piece of code released to the repository (which happens multiple times per day) triggers the execution of all unit tests, which have to pass entirely each and every time before working on the next feature. This rigorous testing regime ensures high-quality code, provides immediate feedback to programmers on their work, and offers crucial support for ongoing software design improvements.

6. **Refactoring**: While focusing on delivering business value in every iteration, XP teams also make continuous design improvements, coined as "refactoring" of code by Martin Fowler [87]. Refactoring includes removing redundancy, eliminating unnecessary functions, increasing code coherency, and, at the same time, decoupling elements.

7. **Pair Programming**: XP proposes that two developers work together on a single computer to write and review code in real time. While it may seem inefficient to have two developers doing the job of one, this practice ensures that all production code is reviewed by at least one other programmer, which results in better design, better testing, and better code [221]. Moreover, pair programming also serves to communicate knowledge throughout the team because, as a result of switching pairs, everyone benefits from everyone's individual and specialized knowledge.

8. **Collective Code Ownership**: This practice declares that code is owned by the team as a whole, and any team member can make changes anywhere in the codebase. The benefit of this is that all code gets the attention of all developers, thus leading to better code quality. Simultaneously, pair programming ensures that knowledge is spread throughout the team so that each developer knows the right place when adding or modifying code for a new feature.

9. **Continuous Integration**: XP proposes that code changes are integrated very frequently, usually multiple times per day. This *continuous integration* ensures that the system is always in a working state and hence prevents code freezes, meaning that developers are blocked and cannot work on important features because of unpredictable problems, which are likely to happen with infrequently integrated code.

10. **40-hour Week**: XP teams maintain a sustainable pace of work, focusing on work-life balance, thus sticking to a maximum of 40-hour work per week and trying to prevent overtime.

11. **On-site Customer**: The customer should fully participate in the development and be present to answer team questions, set priorities, and resolve disputes if necessary.

12. **Coding Standards**: XP teams follow a set of agreed-upon coding standards to maintain consistency and make the codebase easy to understand. As explained by Ron Jeffries, coding standards should result in code that "looks as if it was written by a single – very competent – individual." While "the specifics of the standard are not important," it is more important "that all the code looks familiar, in support of collective ownership" [221].

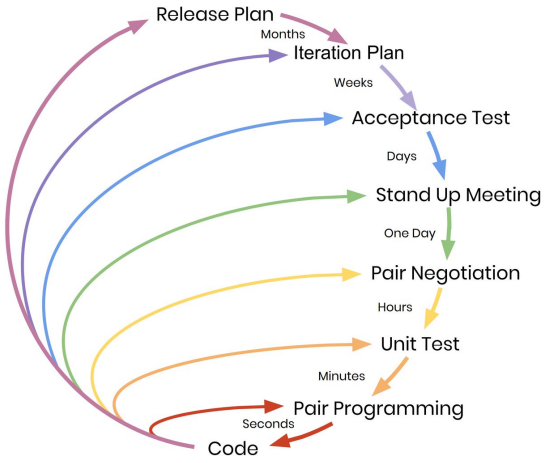


Figure 3.16: XP feedback loop[7]

7 Image based on work by Don Wells:
http://www.extremeprogramming.org/map/loops.html

Given these values and practices, XP does not have a formally defined structure for the software development lifecycle. Instead, it is more concerned with how teams should work together and how tasks should be approached on a day-to-day basis, thus focusing on frequent releases, continuous feedback, and embracing changes.

Before investigating Scrum as the most prominent agile development framework in Chapter 4, the following section elaborates how representatives of the previously presented "lightweight" approaches coined "agile" as a new development paradigm based on four firmly anchored values and twelve common development principles, altogether forming the "Agile Manifesto."

3.6.4 *The Agile Manifesto*

The sections above illustrate a small selection of "lightweight" approaches published in the 1990s. Although some of the inventors were competitors to each other, they soon realized that all of them shared a common vision about a new way of developing software based on an equal set of values and principles.

Therefore, seventeen leaders and representatives of *Scrum* [234], *Extreme Programming* [16], *DSDM* [256], *Adaptive Software Development* [116], *Crystal* [53], *Feature Driven Development* [202], *Pragmatic Programming* [119] and others managed to meet and discuss their approaches as alternatives to the existent heavyweight, document-driven software development process.

The four values of the Agile Manifesto

In a conference-type atmosphere, this group of independent thinkers shared their ideas and very soon decided to favor the term "agile" instead of "light" or "lightweight" as a classification of their approaches to give more meaning to their mutual philosophy and to convey the essence of what these methodologies were about. Despite initial concerns of some representatives that this group of competitors would never agree on anything substantive, the meeting revealed a deep connection between its members, and they soon agreed on a set of compatible values classified as essential for developing software the agile way. These fundamental values were jointly signed in the form of the "Manifesto for Agile Software Development," often shortly referred to as the *Agile Manifesto* [19].

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The twelve principles of agile development

As a complement to these four values, the authors also added "twelve principles of agile development" in order to elaborate on consequences, i.e., what it means for the development process if these values are thoughtfully implemented:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for a shorter timescale.
4. Business people and developers must work together daily and throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

When the manifesto was published in 2001, it struck a nerve and received much attention in the software development community. That is because it provided a *mission statement* and concentrated the individual ideas of former lightweight approaches into one formulated document conveying the foundation of agile development philosophy. By this, together with an article that was published in the "Software Development Magazine" [86], which provided comments about the manifesto by two of its authors (Martin Fowler and Jim Highsmith), the ongoing movement, facilitating change over the attempt to prevent it, gained huge momentum.

Initiated by the paradigm shift of IID (see Section 3.5), realizing that planning and development cannot be a one-shot process in turbulent environments, the new development philosophy finally found its name: "agile."

While the manifesto authors did not claim leadership of the agile community itself and treated themselves as just the ones who helped to launch the ship, some of these authors later actively contributed to a newly formed "Agile Alliance," which is a non-profit group of researchers and experts "promoting the concepts of agile software development as outlined in the agile manifesto" [5].

Until today, the manifesto, with its four values and twelve principles, must be treated as the essence of what agile development is about. It is the foundation and solid ground upon which individual methodologies or frameworks, such as XP, DSDM, or Scrum, can be built. This should be stressed in particular because, as will be explained in Part II of this thesis, many of the existing problems when the industry is trying to adopt agile development arise from the fact that the solid foundation with its core values and principles is often not taken into account or taken seriously enough.

3.7 SUMMARY

As outlined in the chapter, the evolution of SDLC models reflects a significant shift in software development practices over time.

This evolution can be concluded as follows:

- **From Rigidity to Flexibility**: Early models like the Waterfall and V-Model represented a rigid, linear approach to software development, emphasizing strict phases and a lack of iteration. Over time, the recognition of the limitations of these models, especially in handling changing requirements and uncertainties, led to the development of more flexible approaches.
- **Emergence of Iterative Models**: The Spiral Model marked a transition by introducing iteration and risk management into the development process. This paved the way for Iterative and Incremental Development (IID), which further broke down the rigidity of traditional models, allowing for repeated cycles of development and enabling more adaptability and responsiveness to change.
- **Transformation Through Agile**: The advent of agile methodologies like Rapid Application Development (RAD), Dynamic Systems Development Method (DSDM), and Extreme Programming (XP) brought a profound transformation. These methodologies

emphasized rapid prototyping, continuous customer feedback, and adaptive planning. They focused on collaboration, customer satisfaction, and efficient response to change, contrasting sharply with earlier models' highly structured and documentation-heavy approaches.

- **Consolidation in the Agile Manifesto**: The culmination of this evolutionary journey is seen in the Agile Manifesto, which distilled the core values and principles underpinning agile methodologies. This represented a paradigm shift in software development, moving away from conventional, plan-driven approaches to a more dynamic, user-centered methodology that values individuals and their interactions and favors working software over comprehensive documentation, thus necessitating customer collaboration and embracing changes.

In conclusion, the evolution of SDLC models reflects a broader industry trend towards more dynamic, flexible, and collaborative approaches to software development. This shift acknowledges the complex, changeable nature of software projects and prioritizes adaptability, customer involvement, and team collaboration over rigid planning and strict adherence to predefined processes.

With Scrum being at the forefront of all agile development methodologies, the following chapter will now elaborate on all its details.

4

SCRUM: THEORY AND PRACTICE

Originating as one of the lightweight methodologies from the 1990s opposing the traditional way of software development, Scrum began its triumphant advance together with the agile movement and until today became the most frequently used and widely accepted software development process framework (see Figure 4.1).

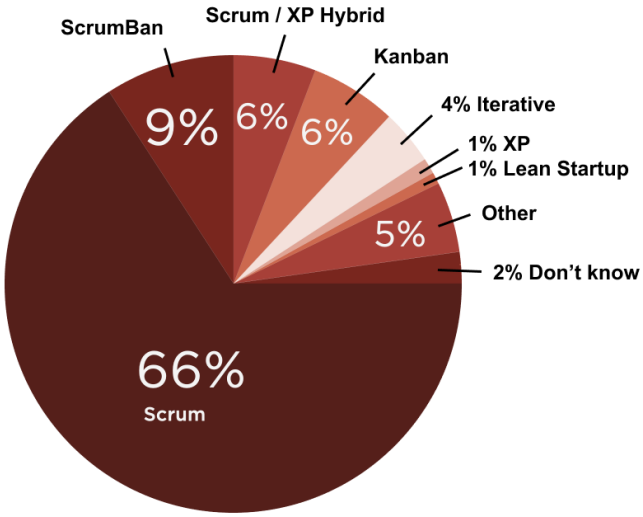


Figure 4.1: Distribution of agile approaches[1]

However, as the second part of this thesis will show, applying Scrum does not necessarily mean implementing it correctly. Too often, teams ignore essential parts of the framework due to environmental restrictions or limited social interactions and, therefore, cannot reach their full potential.

For that reason, the overall goal of this thesis is to contribute to the mastering of Scrum by providing an environment and novel software solution, which is presented in the third and final part and is aimed at helping teams stick to the principles and practices that have proven to be essential, by either the official definition of Scrum itself or leading experts in the field. Beforehand, this chapter lays the foundation by presenting the very details of Scrum, its rules, and best practices to follow. It thus defines the *ideal situation* serving as the target goal of the later provided solution.

1 Image based on the "15th State of Agile Report" [299]

4.1 ROOTS AND SCRUM THEORY

The roots of Scrum can be traced back to the milestone article "The New New Product Development Game," written by Takeuchi and Nonaka for the Harvard Business Review magazine in 1986 [262]. These two organization theorists observed that several companies from Japan and the United States at that time, such as Fuji-Xerox, Honda, 3M, Hewlett-Packard, and others, started to establish new approaches for their product development processes. Using analogies from sports, Takeuchi and Nonaka compared the traditional, sequential development process to a "relay race," in which the baton is passed from one product development phase to the next, which may not be optimized for maximum speed and flexibility. In contrast, the companies mentioned above seem to use a novel "rugby" method, "where a team tries to go the distance as a unit, passing the ball back and forth" [262].

In their article, Takeuchi and Nonaka identified several characteristics of this new approach, including "built-in instability" or "self-organizing project teams," and laid the theoretical foundation for a new way of working that better suits the requirements for a fast-paced, competitive product development process.

Some years later, *Ken Schwaber* and *Jeff Sutherland* adapted their ideas to the domain of software development and introduced *Scrum* as a new process framework in 1997 [234]. The connection to the article by Takeuchi and Nonaka is shown by the chosen name "Scrum," which is also derived from rugby, where the purpose of a scrum is "to restart a play quickly, safely and fairly, after a minor infringement or a stoppage" [217].

The Scrum Guide is the official reference and was released in 2010 by Schwaber and Sutherland

While Scrum slowly enjoyed popularity after its introduction in 1997, the inventors decided to distill the framework into a freely available document known as the *Scrum Guide*, which was released in 2010 and provided an official reference that has been deliberately kept short in order to spread knowledge quickly by avoiding any filling material and only focusing on the essential aspects of the framework.

Since its introduction, the Scrum Guide has been revised six times. In its latest version, Schwaber and Sutherland define Scrum as:

> *"a framework within which people can address complex adaptive problems, while productively and creatively delivering products of highest possible value." [237]*

By this definition, the originators indicate two aspects. At first, the definition opens up a broader perspective by suggesting Scrum for "complex problems" in general, which implies that it may not be

limited to the domain of software development but could be applied to other areas as well.

Second, the creators also emphasize Scrum as a *framework* and not as a "process," "technique," or "method." That is because the Scrum Guide is not strict in employing the process since it neither prescribes concrete techniques nor specific tactics for using the framework. Instead, it states these "vary and are described elsewhere" [237].

Besides being a framework, Scrum is very strict regarding its roles, artifacts, and events

This definition might be very broad and seems unsharp at first glance. However, the specification of the core development process is strict in two respects. At first, it is defined by rules describing an interplay of specific roles, artifacts, and events, which will be considered in the following sections. Second, it stresses the importance of its theoretical foundation based on interlinking concepts from iterative and incremental software development, as presented in Section 3.5, with elements from the *empirical process control theory*, which is shortly introduced in the following.

The defined process control model of sequential approaches

In traditional sequential methodologies, development follows a *defined process control model*. The same outputs are generated every time given a well-defined set of inputs, i.e., a defined deterministic process run until completion always leads to the same results [236]. As explained in Chapter 3.4.1, this theoretical model matches a sequential development process that originates in industrial manufacturing and could be easily adapted to the construction of hardware, which follows a series of well-defined steps. It represents a *plan-based* approach, where the blueprints are fixed and will not change during construction. As a consequence of this entirely determined pre-ahead planning, the outcome is also predictable regarding costs and schedule. However, as shown in previous chapters, the assumption that requirements are fixed and will not change is rarely valid for software projects due to the high complexity and unpredictable side effects of implemented features. When sticking to a defined process control model, a change of requirements during the process affects the original predetermined plan. As a result, the budget and schedule must be changed, too, in terms of higher costs and a later completion date. However, this often becomes impossible due to contractual agreements, so teams still must deliver by the date they initially committed. As written before, this is exactly when projects start to go out of control and are likely to fail.

On the contrary, the agile approach establishes an opposite mindset by assuming that the upfront requirements cannot be understood entirely in advance. Moreover, these initial requirements will not even be fixed since no one knows to what degree future customer feedback will lead to new decisions, which in turn may completely alter the initial requirements set. In fact, the agile approach *welcomes* changes - even late in the development process - as something positive, ensuring that the correct product is being built. Nonetheless, the agile mindset

prioritizes delivering a project by a certain date but allows for flexibility in requirements, as these are expected to evolve. Using value-driven development, i.e., prioritizing and completing those requirements that provide the most value to the customer, agile approaches accept that not all requirements may be finished by the delivery date but emphasize the importance of delivering enough key features to create a valuable and functional system for the customer.

Scrum is based on the three pillars of empirical process control theory

Therefore, progress cannot be measured against a checklist of predetermined requirements but by mechanisms of *empirical process control*, assuming that the only reliable knowledge in highly unpredictable environments comes from observation and experience. Using empiricism, therefore, means to measure progress in a fact-based and evidence-based manner. Scrum ensures this by interlinking many of its elements to the three pillars of empirical process control, which are *transparency*, *inspection*, and *adaptation*, which are building on one another.

Transparency

Transparency means presenting the facts as is. Any parts of the process that affect the outcome are visible and known to everyone in response to the outcome. This includes everybody's ability to observe progress and the project's current state, which also requires that all people involved share a common language and a mutual understanding of what is being seen by defined standards. Scrum ensures these aspects through certain artifacts and progress definitions, acting as information radiators for the whole team by visualizing the state of project planning and progress. As will be explained in the following chapters, these are the *product backlog* and the *definition of ready* (see Section 4.5), the *sprint backlog* (see Section 4.6), and the *definition of done* (see Section 4.3).

However, in Scrum, transparency means more than just visualizing everybody's work status to everyone else. It also means incorporating a work setting of mutual trust and courage to keep each other abreast of good and bad news, which is why no one has any hidden agenda. This is particularly addressed by the *five values* of a Scrum team, which will be presented in Section 4.4.1.

Inspection and Adaptation

When a process becomes transparent, *inspection* is meant to maximize value and control risk by analyzing progress in detecting undesirable variances when heading towards a specific goal. In terms of Scrum, the development process is not inspected by external auditors but by everyone in the Scrum team, which is continuously heading towards improvement and *adapts* to the inspection results.

For example, the team closely collaborates with the customer to gather constant feedback on whether the outcome meets the customer's expectations. This inspection is treated as an opportunity to clarify the requirements to guarantee that the product will be accept-

able. If some aspects deviate outside acceptable limits, everybody will adjust the process as soon as possible to minimize further deviations that otherwise would build up quickly.

In Scrum, inspection and adaption go hand in hand with one another and are subject to four formal events, of which three are concerned with ensuring that *what* is being built is acceptable. These events are *sprint planning* (see Section 4.7.1), the *daily Scrum* (see Section 4.7.2), and the *sprint review* (see Section 4.7.3). In addition, there is one event in particular for inspecting the internal development process and *how* the group performs as a team, aiming at generating insights and strategies for adapting to identified hindrances to guarantee continuous improvement. It is called the *sprint retrospective* and will be explained in Section 4.7.4.

4.2 OVERVIEW OF THE SCRUM FRAMEWORK

The Scrum framework consists of specific parts, all of which interplay with each other by various rules. Since it is difficult to consider each part separately without referencing its related parts, this section gives an introductory overview of the framework structure. This should help establish an approximate idea of all individual aspects before elaborating on details in later sections. If not otherwise stated, all the following information is based on the official Scrum Guide [237].

As Scrum is based on iterative and incremental approaches (see Chapter 3.5), the overall development process is characterized by succeeding iteration cycles, at which end must be an increment to the product which fulfills the requirement to be potentially releasable. Such a development cycle is called *sprint* and functions as a superordinate structure to the interplay of the remaining Scrum components: *roles*, *meetings*, and *artifacts*, as illustrated in Table 4.1.

Roles

Scrum is designed around a self-organizing team composed of three roles. As the name already suggests, the *product owner* solely takes responsibility for managing the *product* by deciding what to build and in which order. His main concern and artifact to work with is the *product backlog*, an ordered list of informal items representing requirements, features, ideas, or other aspects relating to the product. On the other hand, the *Scrum master* is in charge of the *process* by removing impediments to development and guiding the whole team to ensure everybody sticks to the values, principles, and rules. The final role is represented by the *development team*, defined as a small cross-functional collection of professionals with all necessary skills to turn the requirements defined by the product owner into a working product.

Roles	Product Owner Scrum Master Development Team
Meetings	Sprint Planning Daily Scrum Sprint Review Sprint Retrospective
Artifacts	Product Backlog Sprint Backlog Increment

Table 4.1: Scrum components

In addition to these roles, Scrum also requires specific *meetings*, which take place at various points in time of a sprint and which shall ensure that the aspects of empirical process control (see Section 4.1) are put into practice.

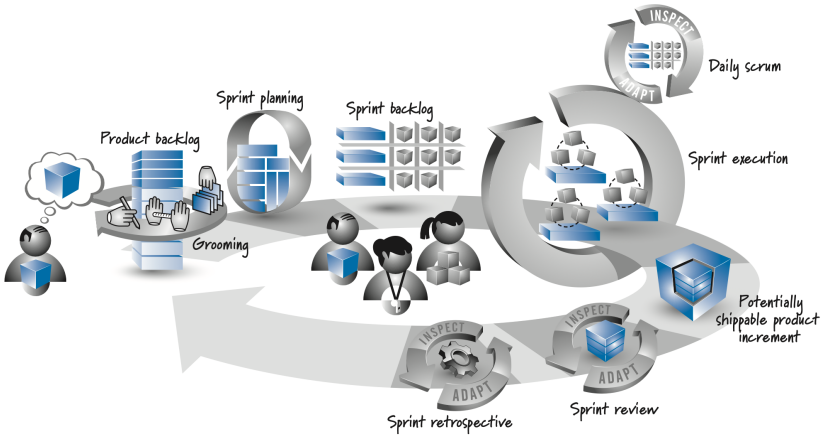


Figure 4.2: The sprint cycle[2]

Meetings and artifacts

As illustrated in Figure 4.2, each sprint begins with the *sprint planning* meeting, in which the team collectively defines a *sprint goal* and agrees to a set of items from the product backlog to be implemented during the current sprint. This selection represents the scope of work and is organized by the *sprint backlog*. During the execution of the sprint, the development team is self-organizing its way of working to finally deliver an *increment* at the end of the sprint that fulfills the requirements as specified before in the sprint backlog. For every member of the development team, it is obligatory to attend the *daily Scrum* meeting, which, as the name suggests, takes place every day as a short gathering of all developers in order to coordinate their tasks

2 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 17]

and speak about possibly occurring hindrances of their work. In the end, the *sprint review* meeting aims to demonstrate the implemented functionality from the development team to the product owner and invited stakeholders. In addition, all participants collectively think about possible next steps for improving the product in succeeding sprints. Before everything is repeated in future iterations, the whole Scrum team is conducting the *sprint retrospective*, which is a meeting in order to inspect and adapt the process of the recent sprint, providing an opportunity to examine which aspects of the development process went well and where is room for improvement.

While this is the basic underlying structure of the Scrum framework, the following sections will elaborate on the individual parts in more detail, beginning with the sprint cycle and its underlying rule set.

4.3 SPRINT CYCLE RULES

The *sprint* as the key Scrum component reflecting the cyclic iterative and incremental development process represents a timebox of one month or less with the fundamental purpose of creating a "potentially releasable product increment" [237]. While the term "increment" will be described in the succeeding section, this section proceeds with the underlying rule set of a sprint, which is a fundamental prerequisite for implementing Scrum correctly.

A sprint represents a timeboxed iteration of one month or less

The timebox of a sprint is limited to one month or less, which ensures that the overall risk for failing is also limited to one month of cost. As an explanation, the Scrum Guide mentions that longer horizons lead to a rise in complexity since the requirements are more likely to change, which increases the risk of false developments [237].

This rule of a short sprint length has a clear connection to the problems of sequential development models, as described in Chapter 3.4. A sequential or "one-way" development method needs perfect upfront planning and specification of the requirements. However, this is conflicting with the real world because of what Boehm already pointed out in 1981 [28] by what Steve McConnell [174] later coined as the *cone of uncertainty* (see Figure 4.3).

This uncertainty is especially problematic for longer projects with a more extensive set of requirement specifications because, for a larger set, the amount of uncertainty has more impact, and the consequences are more serious. For this reason, Scrum treats every sprint as a self-contained project, including a transparent and formulated goal, and thus divides the whole project effort into the work of several sprints, where the risk of failing is limited to one month.

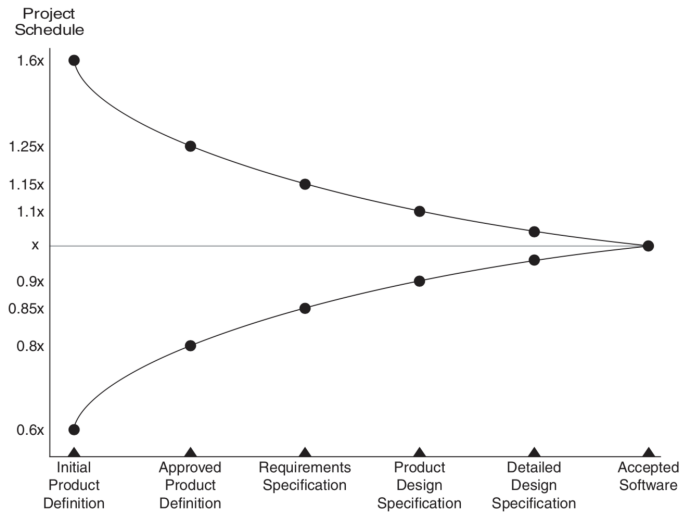


Figure 4.3: The cone of uncertainty narrows as the project progresses[3]

While one month is the maximum, there is no official lower bound for the length of the timebox and no rules for determining the optimum sprint length. However, this decision should not be taken lightly since the timebox determines what Mitch Lacey calls the "stimulus-to-response cycle." He identified three factors that should be considered for determining the best sprint length: the expected duration of the overall project, the customers and stakeholders, and the Scrum team itself [152, p. 80]. In his book "The Scrum Field Guide," he offers a recommender system in the form of a quiz, in which a person must select answers to several considerations for every factor. Based on a scoring key, these answers are translated into point values so that finally, the sprint length with the highest value is recommended as the most suitable for balancing the project duration against the amount of risk someone is willing to take, the ability of the team and the tolerance of the customer [152, p. 87].

The sprint has a consistent timebox

The decision about the length of the timebox is also critical because sprints are also specified to have *consistent* durations. However, this rule of consistency is not as strict as the timebox limitation. Nevertheless, it needs very good reasons to change the sprint duration because of the constant pace of the team. The Scrum Guide also regulates that a new sprint must contain all of the obligatory ceremonies (sprint planning, daily Scrums, sprint review, and sprint retrospective) and that it starts immediately after the conclusion of the previous one.

A sprint can only be canceled if its goal becomes obsolete

Furthermore, a sprint is only allowed to be canceled if - and this is the only exception - the goal of the sprint has become obsolete in the meantime, for instance, due to changes in the market or technological conditions. But overall, sprint cancellation is described as traumatic to the Scrum team and should be very uncommon [237].

3 Source: "Agile Estimation and Planning" by Mike Cohn [56]

Every sprint must fulfill a concrete goal, and it is not allowed to make any changes that would endanger reaching this goal by the end of the sprint

When a sprint is started, it not only consists of a set of features that have been determined to be of the highest value to the customer and hence must be implemented during the iteration, but it also defines a concrete *goal*. This goal ensures that the whole team knows why it is worthwhile to run the sprint and what should be achieved on a higher level [209]. While goal is what really drives the sprint work, it may be possible to re-negotiate the sprint's scope because of new knowledge. However, it would not be allowed to apply changes that endanger meeting the sprint goal since this would affect the quality of the increment to be delivered.

A typical example is a sprint that starts deviating from what has been planned because of unforeseeable problems or new knowledge that has been gained in the meantime. In a well-planned sprint, there should be room to drop particular features that do not directly contribute to the overall goal. That way, the team can still meet the sprint's overall goal and deliver a high-quality increment, while features left open can be addressed in later iterations.

In this context of Scrum, it is important to know that the term "increment" must not be equated with the piecewise delivery of a software product of the purely "incremental development" approach, as explained on Page 46. That is because Scrum incorporates both incremental *and* iterative development. Therefore, "increment" refers to a piece of software that builds up on previously delivered code, including *new* but also *refined* functionality of previous sprints.

For this reason, the Scrum Guide defines it as a "step toward a vision or goal" and as "the sum of all the Product Backlog items completed during a Sprint and the value of the increments of all previous Sprints" [237].

The increment must be "potentially releasable"

The characteristic of the increment to be "potentially releasable" is of utmost importance and, according to Cohn, one of the biggest challenges for a Scrum team to achieve, but at the same time essential for becoming agile. The term actually covers several aspects, as explained in the following.

At first, it refers to "working software" - one of the four critical agile values declared by the Agile Manifesto (see Chapter 3.6.4) - which, according to Cohn, allows valuable "hands-on" customer feedback, helps to avoid unfinished pieces of work for better measuring of the progress and allows the product to be shipped at any time (maybe with fewer features) in order to quickly respond to changes of the competitive market [57, p. 258]. On the other hand, the word "potentially" indicates that the increment does not need to be "truly" releasable. Cohn explains this distinction by an example of a customer who wants to add the functionality of print and print preview to his product. The Scrum team may decide first to implement the print preview in

one sprint, followed by the functionality to initiate and perform the printing in the succeeding sprint. In that case, a release to the customer after the first sprint would not be reasonable because although the print preview is solid and usable and hence could potentially be delivered, the outcome is not cohesive because of the strong relationship to the second sprint [57, p. 260].

The definition of done determines conditions for being potentially releasable

Another aspect is covered by Rubin, who noticed that "potentially releasable" is also a "state of confidence" that the work of a sprint is really done, so there is nothing crucial left over when the business decision to ship the results of the sprint is taken [226, p. 74]. In order to determine whether the increment has reached the state of being potentially releasable, the team must have agreed upon a well-defined *definition of done* (DOD), which is a checklist of conditions that must be satisfied to declare the increment as being "done." For example, conditions could be that all features are coded, designed according to a set of style guides, well-written, integrated into a mutual code base, and verified by succeeding test mechanisms.

These conditions of satisfaction are transparent to all team members and may evolve over time as the team gains more knowledge, thus making the definition of done subject to inspection and adaptation as elements of the empirical process control.

4.4 THE SCRUM TEAM

According to the role model of Scrum, a team consists of a *product owner*, a *Scrum master*, and the *development team*. Before investigating these individual roles, the following section elaborates on the team size and five essential values contributing to solid team cohesion and mandatory spirit when working in turbulent, fast-paced agile environments.

4.4.1 *Five Values and Team Size*

A team consists of 3-9 developers, one product owner, and one Scrum master

In terms of the overall team, Scrum stipulates two rules. The first is about the team size, which should be small and consist of no more than three to nine people forming the development team plus one person, each occupying the role of the product owner and Scrum master. The second rule is about five values or character traits every member of the Scrum team should have internalized: commitment, courage, focus, openness, and respect (see Figure 4.4).

Figure 4.4: Scrum values[4]

Each team member must work according to five values

Small team size is necessary to establish a working environment in which all members *commit* to achieving the goals as a group, while each individual member needs to have *courage* to work on challenging problems and must preserve *focus* on the work and goals of the whole team. Being *open* about tasks, goals, and challenges is crucial for establishing a transparent process, which holds not only for the Scrum team itself but for the stakeholders as well. Team members must *respect* each other, being capable, independent people with individual strengths and weaknesses. According to the inventors, only by living these values, "the Scrum pillars of transparency, inspection, and adaptation come to life" and build trust for everyone [237].

In the case of a bigger team size, the originators state that complexity would increase, and sticking to the values mentioned above would be harder to guarantee. As indicated by the values, Scrum focuses on professionals working in a corporate culture, inspiring an atmosphere that promotes team spirit, mutual cooperation, and willingness to help each other. Although differentiated by the already mentioned role concept, they mutually share responsibility for the outcome of the product development process and make decisions on their own without being driven from the outside, which is why the team is limited in size to function as one unit.

In particular, the size of the development team is limited to 3-9 persons to be "small enough to remain nimble and large enough to complete significant work within a Sprint" [237]. According to Schwaber and Sutherland, smaller teams (< 3 persons) could be unable to deliver an increment of high quality because of skill constraints. In contrast, bigger teams (> 9 persons) would need too much coordination and hence exceed the useful amount of complexity for an empirical process.

4 Source: https://www.scrum.org/resources/scrum-values-poster

Although not explicitly mentioned, this thought builds upon the outcome of the famous book "The Mythical Man-Month," published in 1975 by Frederick Brooks [39], also known as "Brooks's Law":

> *"Adding manpower to a late software project makes it later."*
>
> — Frederick Brooks [39]

The rationale behind this is that the overhead of communication and coordination tends to rise as the square of the number of communication paths between developers, or more precisely by the following formula, which is illustrated in Figure 4.5:

$$Number\ of\ communication\ paths = \frac{n(n-1)}{2}$$

where n is equal to the number of developers.

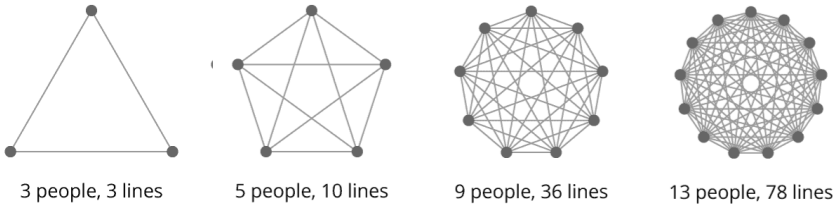


Figure 4.5: Brooks's law: people and resulting lines of communication

For that reason, and since face-to-face communication is considered to be "the most efficient and effective method of conveying information to and within a development team," as stated in the sixth principle of the Agile Manifesto (see Chapter 3.6.4), more developers would not scale to comply with the conditions of a transparent process.

But what if software projects are bigger in size and cannot be accomplished with a single Scrum team? Moreover, how may the Scrum philosophy that works well for one team be applied to the organization as a whole? These are questions concerning the *scaling* of Scrum, which are out of the scope of this thesis, focusing on the challenges and issues of single Scrum teams and Scrum in general. It should, therefore, only be said at this point that various scaling models exist, e.g., *LeSS*, developed by Bas Vodde and Craig Larman [264], *Safe*, developed by Dean Leffingwell [230], *Scrum@Scale*, developed by the co-creator of Scrum Jeff Sutherland [242], and *Nexus*, developed by the co-creator of Scrum Ken Schwaber [248], which each extend the Scrum framework by their own rules about the interaction of several teams to work from a single product backlog and build an integrated increment at the end of each sprint.

4.4.2 *The Product Owner*

The product owner decides what to build

The *product owner* is one dedicated person, and his first and foremost task is to *maximize the value* of the product by deciding what to build and in which order [237]. He, therefore, is working in close collaboration with external (e.g., clients or customers) and internal stakeholders (e.g., marketing or sales division) to distill a shared vision of the product and constantly gather new information for updating the requirements that are specified in the product backlog, for which he is solely responsible. In addition, he represents the link between the stakeholders and the development team (see Figure 4.6) by assuring a proper *product backlog management* (see Section 4.5) so that everybody understands the stakeholders' needs (specified as requirements in the product backlog), their priority (reflected in the ordering of the product backlog) and works towards the right goal.

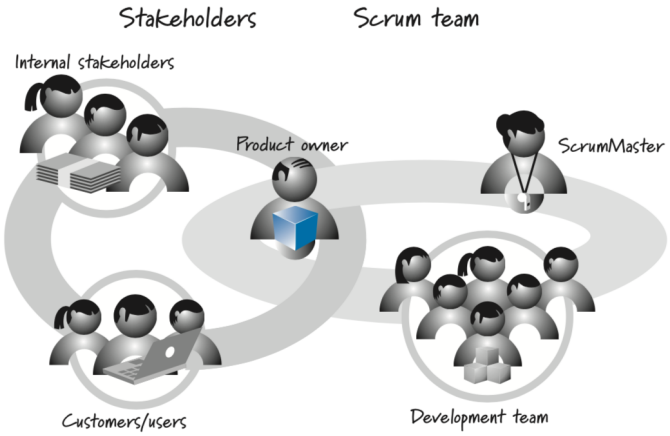


Figure 4.6: The product owner role[5]

By this, the product owner becomes the "empowered central point of product leadership" [226, p. 165]. Being close to the business side, he defines and priorities the features to be developed and also accepts or rejects the work results, resulting in responsibility for the profitability of the product and the stakeholders' return on investment by constantly comparing the vision against the reality and making tradeoffs between scope, budget, and quality [57, p. 125].

4.4.3 *The Development Team*

The development team consists of 3-9 people who are solely responsible for turning the workload of the sprint (as defined by the sprint backlog, see Section 4.6) into a "potentially releasable increment." While the sprint backlog results from the product owner's decisions

5 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 165]

about *what* to build according to his priorities, it is entirely up to the development team *how* to achieve this. For this, the Scrum Guide defines the development team to be *cross-functional* and *self-organizing*.

Developers are cross-functional, self-organizing and decide how to turn the sprint backlog into a releasable product increment

Cross-functional means that *all* team members are equipped with the necessary skills to create the increment. Scrum is against using specific titles for individual team members (e.g., "architect," "tester," "UX designer," etc.) and requests to omit sub-teams for specific domains of the development lifecycle. This is fundamentally different from traditional approaches, where individual job roles are typically assigned to different people, so teams are apparently formed by several experts occupying specialized, role-specific positions. For example, a team of software architects could be responsible for modeling and structuring the application, whereas a team of programmers implements features that are subsequently validated by another team of testers. Scrum has firmly discarded this notion of developer roles and role-specific teams since there are neither titles nor sub-teams in the development team. So, instead of working on "horizontal slices" of the product, each member of the development team is considered equal, regardless of his or her individual experience and specialty, and therefore works on "vertical slices" of the product, i.e., each member designs, implements, integrates, and tests end-to-end functionality of the increment [226, p. 195].

However, this cross-functionality is not meant to exclude members with specialized skills or focus areas [7]. In fact, Rubin mentions that it would be unrealistic to assume that each developer is equally good at everything and therefore proposes that members of the development team should have both broad and individual "deep" skills, which he refers to as a "T-shaped skill set" [226, p. 201]. The "deep" skills are individual focus areas and specialties, which allow developers to guide the rest of the team in particular aspects, whereas a broad skill set not only allows them to participate in every part of the implementation process but also leads to additional flexibility of the team, because of overlapping knowledge. If the team is, for instance, behind testing, a developer with less profound testing skills could still support developers with more sophisticated testing skills to overcome development bottlenecks [226, p. 201].

The example above also shows the second characteristic of the development team, which is the request for self-organization. It allows developers to organize and manage their own work and autonomously determine the best way how to achieve their goals and turn requirements into releasable functionality. No one outside the development team, neither the product owner nor the Scrum master, should be in a position to exert influence. Therefore, it is up to the Scrum master to protect the development team against outside influences in order to optimize its efficiency and effectiveness, which is in accordance with

the eleventh principle from the Agile Manifesto stating that "the best architectures, requirements, and designs emerge from self-organizing teams" [19] (see Chapter 3.6.4).

With these two characteristics of the development team, Scrum serves two purposes. At first, cross-functionality and no titles encourage that accountability of the outcome always belongs to the development team as a whole, making all team members responsible for both the success and the failure of sub-systems or the entire system [52]. Thus, it prevents blaming, finger-pointing, and too much time spent complaining when something goes wrong [226, p. 171]. Second, self-organization targets close collaboration between team members. The former stereotype developer from traditional approaches, isolated by others and without the urge to talk to anyone else for the rest of the day, does not apply to Scrum. Instead, developers actively participate in the various Scrum activities and constantly inspect and adapt their development process [57, p. 177].

4.4.4 *The Scrum Master*

The Scrum master constantly enhances the development process according to the Scrum rules

While the product owner is in charge of the product, the Scrum master is responsible for the *process*. He has to ensure that the Scrum theory, rules, practices, and values are understood, enacted, and enforced among all parties involved. This role represents an important link between the company's management level and the Scrum team by providing services to the organization, the product owner, and the development team as a form of *servant leadership*.

The list of services described by the Scrum Guide is shown in Table 4.2. While some aspects might look simple, such as managing and coordinating the Scrum meetings or collecting status information from the team members, Lacey exhorts the complexity of the work and says there might be a tendency to underestimate it. He describes the Scrum master as a "change agent" with tremendous impact on the attitude and company culture in terms of living the agile philosophy, in which the Scrum master represents "the fluid that ensures the team's gears are turning at optimum effectiveness" [152, p. 71].

With respect to the development team, the Scrum master may be seen as being in a contradictory position since the Scrum Guide describes the role as a servant leader on the one hand, but also someone without authority (since the development team is self-organizing). Cohn clarifies this contradiction by saying that the Scrum master has indeed no authority over the team but authority over the process and compares this role to a "personal trainer" who provides motivation, reminds of the goals, and tries to make people perform at their best [57, p. 118].

Scrum Master Service to the Organization
Leading and coaching the organization in its Scrum adoption
Planning Scrum implementations within the organization
Helping employees and stakeholders understand and enact Scrum and empirical product development
Causing change that increases the productivity of the Scrum Team
Working with other Scrum Masters to increase the effectiveness of the application of Scrum in the organization
Scrum Master Service to the Product Owner
Ensuring that goals, scope, and product domain are understood by everyone on the Scrum Team as well as possible
Finding techniques for effective Product Backlog management
Helping the Scrum Team understand the need for clear and concise Product Backlog items
Understanding product planning in an empirical environment
Ensuring the Product Owner knows how to arrange the Product Backlog to maximize value
Understanding and practicing agility
Facilitating Scrum events as requested or needed
Scrum Master Service to the Development Team
Coaching the Development Team in self-organization and cross-functionality
Helping the Development Team to create high-value products
Removing impediments to the Development Team's progress
Facilitating Scrum events as requested or needed
Coaching the Development Team in organizational environments in which Scrum is not yet fully adopted and understood

Table 4.2: Scrum master services [237]

Many further paraphrases for the Scrum master role can be found, giving strong evidence for its complexity of work (see Figure 4.7). For example, Rubin describes the role as "coach" of the Scrum values and principles as well as "impediment remover" for the development team, taking responsibility for any barriers or problems that would slow down the developing process while at the same time acting as a protection shield for the development team, blocking any interferences from the outside [226, p. 185].

In this sense, the Scrum Master is also a "mediator" between the product owner and the development team, as described by Cohn

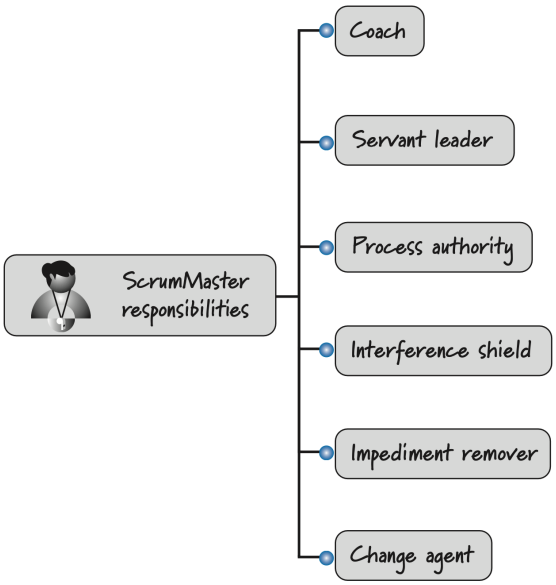


Figure 4.7: The Scrum master role[6]

[57, p. 131]. While the product owner may strive for the maximal workload of the development team to achieve the greatest amount of throughput, the Scrum master is responsible for balancing the product owner's expectations against the health and well-being of the development team. According to Cohn, this is why both roles should be clearly separated and never occupied by the same person, because this positive tension promotes the team's efficiency [57, p. 131].

4.5 THE PRODUCT BACKLOG AND ITS MANAGEMENT

Scrum's most important and exclusive artifact for planning the development process and managing requirements is the *product backlog*. It is defined as "an ordered list of everything that is known to be needed in the product" [237]. While this definition seems rather general, it unifies nearly all critical aspects mentioned in the Scrum Guide, which are investigated in the following.

The product backlog is a prioritized list of everything needed to improve a product

At first, the product backlog is the *"list of everything,"* meaning it is the *single source of requirements*, and the development of every individual product is driven by only *one* product backlog accordingly. "Everything" also indicates that the product backlog does not only contain specific types of elements. An element of the list is referred to as *product backlog item* (PBI) and may represent different types of requirements, such as feature definitions, bug fixes for existing functionality, or visions for future enhancements (see Figure 4.8).

6 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 186]

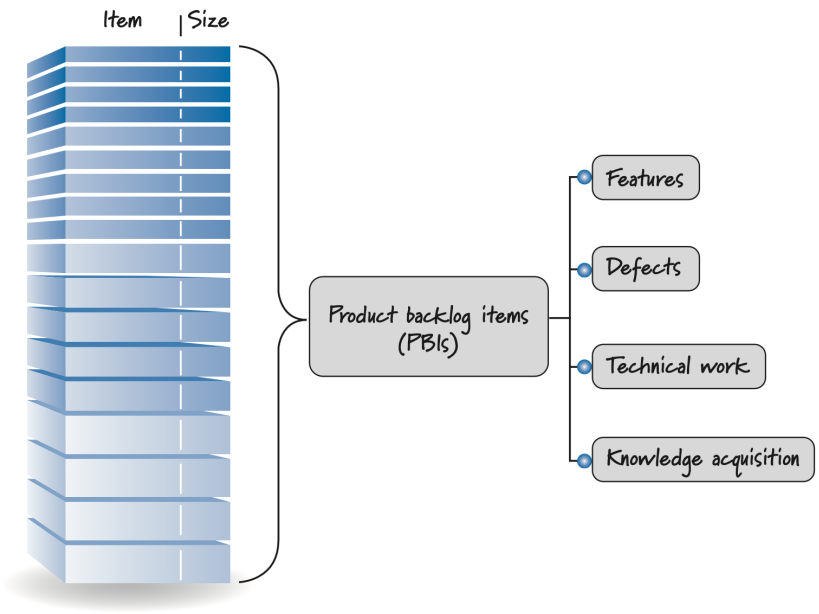


Figure 4.8: Product backlog items[7]

While the Scrum Guide omits concrete recommendations for the appearance and design of a PBI (however, there is a de-facto standard, which is presented in Chapter 4.8.1), it does say that an item must have a description, order, estimation, and (business) value [237].

A good product backlog is "DEEP"

Secondly, the product backlog represents *"everything that is known to be needed,"* which by itself indicates several aspects all at once, which can be addressed by an acronym used by Roman Pichler, who describes a good product backlog as being *DEEP* - detailed appropriately, emergent, estimated, and prioritized [208, p. 48].

To begin with, the phrase above alludes to software development as a process of gaining *knowledge* about the product. Knowledge results from a learning process gained through experience and, therefore, is never complete as long as learning continues. For that reason, a product backlog cannot be declared "complete" as long as the product is being developed or maintained. Instead, it is officially described as a *living artifact*, making it fundamentally different from formal requirements documents of traditional software development models since Scrum acknowledges that it is impossible to gather all requirements correctly upfront and that changes are inevitable. As the project evolves, more knowledge is acquired with every increment delivered and through continuous feedback from the customer.

As a result, new requirements will *emerge* in the form of new PBIs, which is why product backlogs may become very big over time. In addition, knowledge that has been acquired about something that is

7 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 100]

yet not finally existent (the future increment) is also by no means static and cannot be assured (this represents the "evolutionary" aspect of IID, as described in Chapter 3.5.2). Hence, customer feedback can also affect existing requirements, which must be adapted to fit the newly acquired knowledge. For this, the Scrum Guide explicitly mentions *product backlog refinement* as an ongoing process in which the product owner and the development team collaborate on reviewing and revising existing items, which should consume no more than 10% of the development team's capacity.

Product backlog grooming

This progressive refinement or *product backlog grooming*, as it is called by Cohn [59], includes adding details, (re)-estimating the amount of effort, updating the ordering according to the new business value, or even deleting items when it becomes clear that they are not needed anymore (see Figure 4.9).

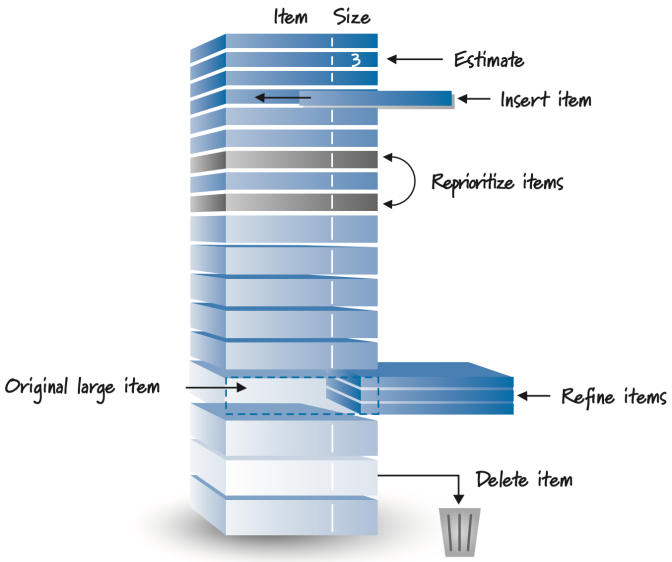


Figure 4.9: Product backlog grooming[8]

The aspect of an emergent product backlog goes hand in hand with the characteristic of PBIs that Pichler describes as *detailed appropriately*. The product backlog is dynamic not only in terms of its size and the number of elements contained but also in terms of the degree of specification of individual items. This means that not all PBIs are specified down to the last detail, which again would be equal to the heavy upfront planning of traditional software development processes and lead to the problems discussed in Chapter 3.4.

Instead, Scrum makes use of an approach that is just-in-time and just-enough. This means the team invests time only when required and plans as little ahead as necessary. As Cohn points out, nearly all projects are time-constrained. Hence, it is wasteful to treat all

8 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 105]

requirements as equivalent when there is actually no need to do so because things are very likely to change, and priorities will shift over the course of a project [57, p. 245]. This likeliness of change is especially true for functionality, which is planned to be implemented in the distant future. It will have many dependencies on previously implemented features, so the specification of details is error-prone. Moreover, it might even be that the functionality becomes obsolete and will not be implemented at all. In both cases, the Scrum team would waste valuable time continuously updating detailed specifications or putting a reasonable amount of work into something unnecessary.

That is why PBIs are big and roughly specified for product functionality that might be implemented in the future, whereas they are small and very detailed for functionality that will certainly be implemented anytime soon. The size of PBIs, "big" and "small," references Pichler's third characteristic of a good product backlog, which is *estimation*. Usually, there is a correlation between the level of specification of a PBI and the estimated effort for implementing its functionality. PBIs with rough specifications have a greater amount of uncertainty. Hence, they are bigger in terms of the estimated effort. On the contrary, when customer feedback or other strategic decisions lead to increasing demand, so the product owner decides to implement the item, it gets progressively refined into several more detailed PBIs, which are smaller and estimated with less effort since they became well-understood. Figure 4.10 shows an estimated product backlog using the "story points" and "T-shirt sizes" metrics, which will be explained in Section 4.8.2.

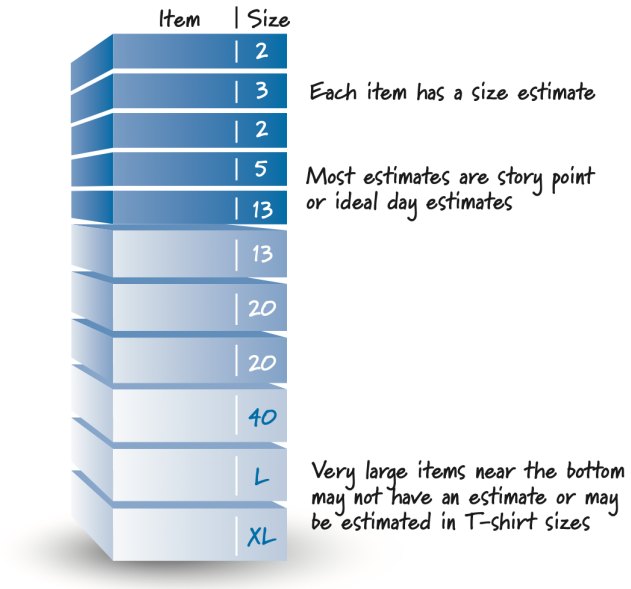


Figure 4.10: Product backlog estimation[9]

9 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 103]

The priority of an item is reflected by its position in the product backlog

Estimation also strongly relates to the *priority* of an item - Pichler's final characteristic of a good product backlog - and the *ordering* requested by the Scrum Guide. As described in Section 4.4.2, the product owner represents a "value maximizer" for the product by deciding *what* to build and in which *order*. This decision is based on the priority of individual PBIs, which should also be reflected in their ordering for maximum transparency across the whole Scrum team.

Therefore, PBIs with a higher priority are located at the top of the product backlog, whereas lower-valued items are at the bottom. Since the product backlog is emergent and depends on customer feedback, the priority of an item itself is constantly re-evaluated - and hence the position in the backlog - and determined by several factors, including the business value, estimated effort reflecting the cost, as well as the knowledge and risk [226, p. 18]. However, Rubin mentions that it is not practical to prioritize every individual PBI. Instead, he suggests an unambiguous priority only for the items planned to be delivered soon. These are well-known and determined, whereas PBIs representing functionality that might be implemented in the future are likely to change (since they are less detailed, as mentioned before). Therefore, a differentiation in priority does not make sense [226, p. 103].

Overall, these DEEP characteristics are strongly connected: PBIs with the highest priority *and* the most detailed specification *and* a smaller estimation of effort, which ensures the practical feasibility within one sprint, are at the top of the product backlog. On the contrary, features with less priority *and* fewer details *and*, therefore more uncertainty, lead to bigger items at the bottom of the product backlog. This ordering, as a result of the DEEP characteristics, ensures that the development team constantly works on the most important features and hence maximizes the value of the product [57, p. 254] by selecting the topmost items first.

The definition of ready determines whether an item can be considered for the next sprint

According to the Scrum Guide, this selection process should be supported by a *definition of ready* (DOR), which represents a checklist with conditions of satisfaction every PBI must fulfill to declare it as "ready" for being added to a sprint [237].

Rubin has given the following DOR as an example [226, p. 110]:

- The business value is clearly articulated
- Details are sufficiently understood by the development team
- The PBI is estimated and small enough to comfortably be completed in one sprint
- Acceptance criteria are clear and testable
- The Scrum team understands how to demonstrate the finished PBI at the sprint review

Notice that PBIs fulfilling this exemplary definition of ready have not been necessarily specified to the furthest extent. Instead, the DOR only ensures that "details are sufficiently understood" so that the development team can be confident in delivering the functionality within the timebox of the actual sprint and commit to the PBI's clearly defined and testable acceptance criteria. With this example, Rubin references the previously mentioned practice of specifying "just enough" and deferring decisions about details to the last responsible moment [226, p. 249]. As a remark, this strategy applies not only to the product owner and the specification of PBIs, as illustrated within this section, but also to the development team and their work management during the sprint, as will be explained in Chapter 4.8.3.1.

According to Rubin, all the aspects mentioned should be considered for a well-managed product backlog as a snapshot of everything known (or, more precisely, "expected") to be needed in the product to fulfill the customer's desires. Thereby, considering the DEEP criteria and a strong definition of ready "will substantially improve the Scrum team's chance of successfully meeting its sprint goal" [226, p. 110].

4.6 THE SPRINT BACKLOG

The sprint backlog results from the sprint planning event (see Figure 4.11) and is defined by the Scrum Guide as "the set of PBIs that have been selected for the actual sprint, plus a plan for delivering the product Increment and realizing the Sprint Goal" [237].

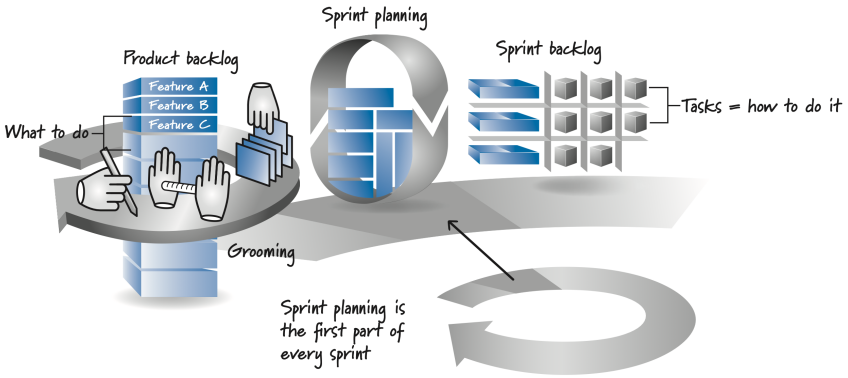


Figure 4.11: The sprint backlog as outcome of the sprint planning event[10]

With this definition, the primary purpose of the sprint backlog is to link *what* should be built during the sprint with strategies *how* to achieve this, thus making decisions and progress transparent to the whole Scrum team.

10 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 21]

The "what" part is the result from the sprint planning event in the form of several items from the product backlog, selected by the highest priority and specified just enough so that the development team was able to commit to being able to deliver them within the timebox of the sprint and under consideration of the definition of done (see Section 4.3). In conjunction with a specified *sprint goal*, the sprint backlog represents a forecast to all stakeholders about what functionality can be expected by the end of the sprint in the form of a potentially releasable increment.

On the other hand, the "how" part is by no means predetermined by the beginning of the sprint. The actual development plan emerges *after* the initial planning event, and it gets continuously adapted throughout the sprint, thus making the sprint backlog a highly dynamic artifact. When the development team learns more about the necessary work and its technical implementation and discusses the progress in their daily Scrum meetings (see Section 4.7.2), the sprint backlog is modified, i.e., concrete *tasks* for the next steps are added, whereas other elements of the development plan are removed, once they are deemed unnecessary.

The sprint backlog is the outcome of the sprint planning event and contains a set of backlog items, the sprint goal, and at least one process improvement

While the Scrum Guide states that the sprint backlog should overall always act as "a highly visible, real-time picture of the work that the Development Team plans to accomplish during the Sprint" [237], there are hardly any rules concerning the management, handling or appearance of the sprint backlog. It is important, though, that it belongs exclusively to the development team. No outstanding person is allowed to add, remove, or modify items from the sprint backlog - not even the product owner or other persons from the management level. This rule guarantees that the development team stays self-organized, which means that all team members mutually have full responsibility for their decisions concerning the implementation of the product and for organizing their work according to their own needs.

Among product-related aspects, the sprint backlog must also contain at least one high-priority process improvement, which results from the sprint retrospective of the previous sprint (see Section 4.7.4). This regulation ensures that process and team improvements become part of daily business and are treated equally important to implementing product functionality.

4.7 THE SCRUM EVENTS

A sprint cycle, as illustrated in Figure 4.2 on Page 78, consists of four events, which will now be explained in chronological order.

4.7.1 *Sprint Planning*

Sprint planning is timeboxed to twice the hours of number of weeks of a sprint

As the name suggests, the *sprint planning* meeting (see Figure 4.12) aims to plan the work for the next sprint. The event itself is timeboxed to a *maximum of 8 hours* for a sprint of one-month length, as specified by the Scrum Guide, or, as suggested by Lacey, *twice the number of hours compared to the number of weeks in a sprint* [152, p. 361]. This timebox ensures focusing on the essentials and avoids too much upfront planning.

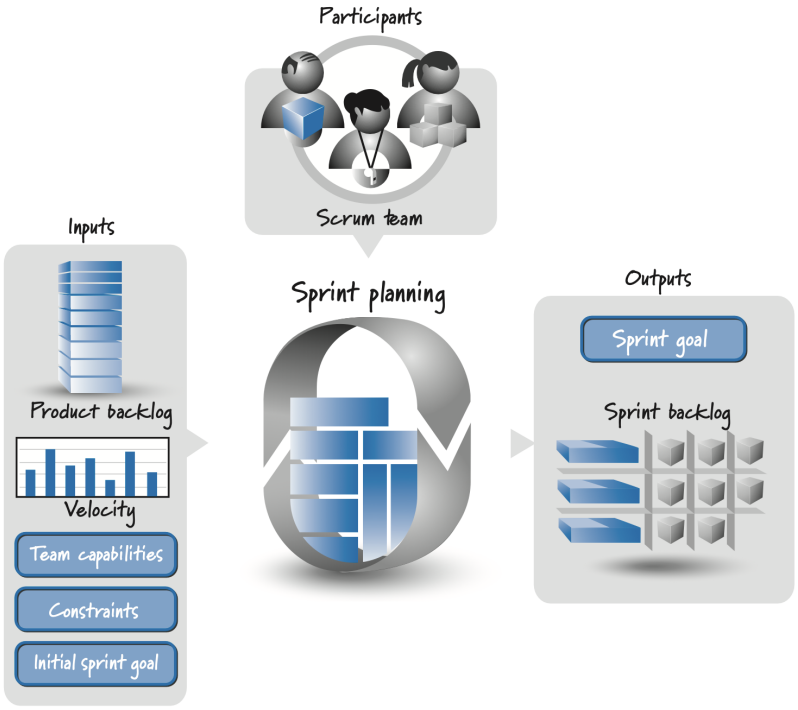


Figure 4.12: Sprint planning[11]

The meeting considers two aspects of planning:

1. *What* can be delivered at the end of the sprint?
2. *How* is this work supposed to be done?

The "what" part of sprint planning

The first part of sprint planning is deciding what can be done during the sprint, and it involves the entire scrum team. Since the product backlog has been ordered by priority (see Section 4.5), the product owner should be able to describe the objective of the present sprint and explain the prepared items he considers to be of highest value to the customer.

After the team established a common understanding, the development team eventually re-estimates PBIs from the prepared set,

11 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 337]

depending on whether an item has changed in the meantime, so the mutual understanding and hence the existing estimation could no longer be guaranteed, or if the item has not yet been estimated at all. In any case, at some point, there must be a well-prepared set of PBIs fulfilling the definition of ready (see Page 4.5) so that the Scrum team can make a joint decision and mutual agreement on which items should be selected for the sprint. For this, it is important to understand that only the development team can assess what can be accomplished during the sprint and hence solely decides on the actual *number* of selected items from the product backlog, taking into account how the development team performed in the past and what the projected capacity is during the present sprint.

The sprint goal provides an objective for the sprint

Besides this selection of items from the product backlog, the what-aspect of the sprint planning activity also includes the collaborative crafting of a *sprint goal*, which is specified as "an objective that will be met within the Sprint through the implementation of the [selection from the] Product Backlog" [237]. By formulating the desired outcome of the sprint, the sprint goal provides *context* for selecting PBIs and, therefore, *guidance* to the development team by giving meaning to the intended outcome.

In addition, it allows a certain amount of flexibility regarding the functionality to be implemented during the sprint. If, for instance, the development team realizes that unpredictable issues lead to more work than initially thought, it has to collaborate with the product owner to negotiate the scope of the sprint. However, as long as the sprint goal can be met, the remaining work is still valuable and target-oriented. The only reason and essential criterion for canceling a sprint is when the sprint goal does *not* provide a target and no longer makes sense, for example, due to superseded strategical decisions or outside circumstances, like changes in the market or advancements in technology. In conclusion, the sprint goal creates alignment among the whole Scrum team, making everyone work towards a common objective and allowing easier communication with stakeholders about what the Scrum team is working on [208, p. 59].

Once the decision about what should be implemented during the sprint is taken, the selected PBIs are transferred to the sprint backlog (see Section 4.6), and the development team creates a plan for how to implement the selected functionality into an increment fulfilling the definition of done (see Section 4.3).

The "how" part of sprint planning

The official specification of Scrum does not provide details on how the "how" should be specified by the development team since this is a matter of self-organization. However, it mentions that the development team designs the system and the work needed to convert the sprint backlog items into a working product increment by decomposing them into smaller units of one working day or less. Due to the general

strategy of just enough upfront planning, this decomposition of tasks is not a one-time task but rather an ongoing activity during the sprint. That is why the "how" part of sprint planning only considers work for the first days of the sprint so that the development team can immediately start to build the increment after the sprint planning is closed.

As can be imagined, specifying the "how" of development includes many technical details. These are of no concern to the product owner, which is why it is common practice to split the entire sprint planning into two succeeding sessions - the first attended by the whole Scrum team to determine the selection of PBIs for the sprint and to craft the sprint goal, the second only attended by the development team and invited other domain experts to decompose selected items into low-level action steps describing development *tasks*.

4.7.2 *The Daily Scrum*

The daily Scrum is an internal developer meeting with a fixed 15 minute timebox

During sprint execution (see Figure 4.13), the *daily Scrum* is the development team's central meeting for applying inspection and adaptation as part of the empirical process control (see Section 4.1). It is meant to be an internal meeting for the developers and targets improving communications, promoting quick decision-making, and reinforcing the team's knowledge level. Independent from the sprint length, the event is time-boxed to *15 minutes* and, as the name suggests, held on a daily basis (see Figure 4.14).

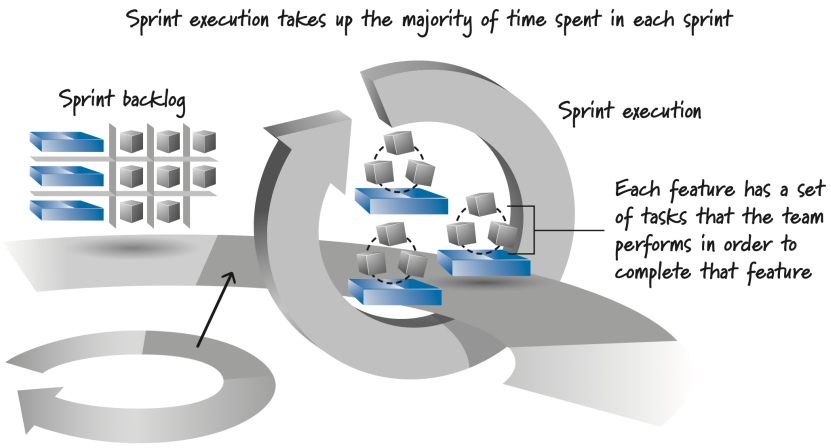


Figure 4.13: Sprint execution[12]

It is also essential that the meeting takes place at the same time and location because this habit reduces complexity since there is no need for invitations, announcements, or extra organizational efforts.

12 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 23]

The main objective of the daily Scrum is to plan the work for the next 24 hours by synchronizing the activities of the development team, identifying impediments, and collectively understanding how the team members are progressing towards completion of the sprint goal [226, p. 354]. For this, the team inspects the achievements since the last daily Scrum, adapts the sprint backlog accordingly, and forecasts what should be implemented until the next day.

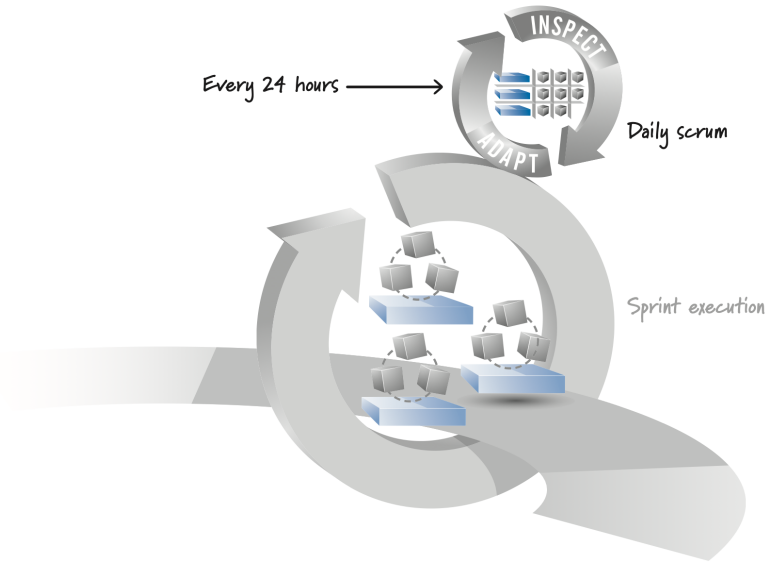


Figure 4.14: Daily Scrum[13]

The three questions of the daily Scrum

The Scrum Guide does not specify details about the structure of the meeting since it is up to the self-organized development team to shape it according to their needs. However, it references an example that has become a prevalent method, which structures the daily Scrum meeting by what is known as "the three questions" each developer must answer one after the other:

- *What did I do yesterday?*
- *What will I do today?*
- *Are there any impediments blocking my way?*

However, this structure should not suggest treating the daily Scrum as the sum of individual status reportings, as noted by Jeff Sutherland, one of the Scrum creators. This would be a bad habit by developers acting as "Scrum Zombies" and not in line with the original intention of Scrum following a rugby approach, where the daily Scrum is closer to a team huddle, full of engagement by motivated players and a solid commitment to a quickly elaborated strategy on how to move forwards towards victory [260]. Therefore, all development team members must

13 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 24]

understand how the daily Scrum aims to live the team spirit and work together to develop a daily strategy for accomplishing the sprint goal. For this reason, the daily Scrum must be short. That way, all participants will stay focused and have the chance to foster the "let's do it" culture collaboratively.

Ensuring that the development team is conducting the daily Scrum and sticks to the limitation of 15 minutes is of great importance and falls within the responsibilities of the Scrum master. He also collects the identified impediments and initiates that problems get resolved as soon as possible.

Usually, the daily Scrum also reveals topics that need cooperation between individual developers, like task overlaps, functional dependencies, or technical challenges, so a person might have to rely on help from another team member. These topics must be addressed separately after the daily Scrum by the persons directly involved since they are of no concern to the team as a whole.

4.7.3 *Sprint Review*

The sprint review is timeboxed to the same number of hours as the number of weeks in a sprint

The *sprint review* is one of two important inspect-and-adapt meetings held at the end of the sprint. It is time-boxed to the *same number of hours as the number of weeks in a sprint* (e.g., 2 hours for a 2-week sprint), which should be sufficient time for the main objective, which is to inspect the work of the current sprint and adapt the future development, as illustrated by Figure 4.15.

Inputs to this meeting are the sprint goal, the sprint backlog, and the outcome of sprint execution, i.e., the potentially releasable increment. On this basis, the increment is inspected and reviewed against what has been specified before. Feedback from all participants of the meeting then leads to a set of identified enhancements and additional features of the product that are reflected in an update (adaptation) of the product backlog [226, p. 369].

To obtain feedback, the sprint review is not only attended by the entire Scrum team to demonstrate the outcome and answer questions but also by other interested parties that have been invited by the product owner to see and discuss the sprint results. These could be internal stakeholders, such as business owners, executives, or managers who are invited to see the progress firsthand in order to suggest course corrections if necessary, or representatives from other internal departments, like sales, marketing, or product support, that might want to provide feedback or ask questions concerning their particular domain. Moreover, external stakeholders could be invited, such as customers,

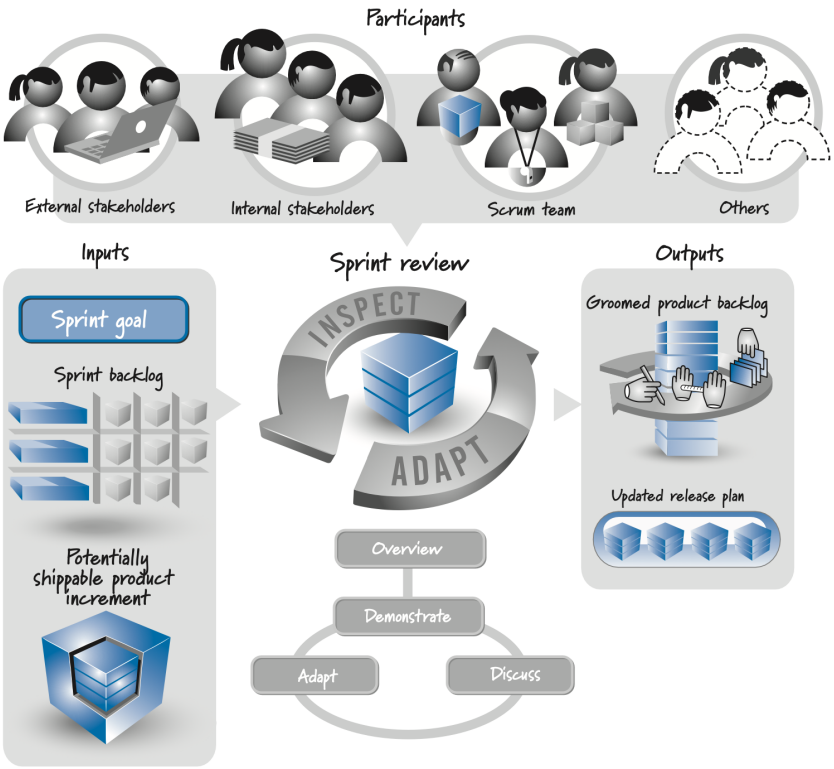


Figure 4.15: Sprint review[14]

users of the final product, or key partners that could provide valuable feedback for future development [226, p. 364].

The sprint review is all about collecting feedback on how to improve the product

The Scrum Guide explicitly mentions that the sprint review is informal and not a status report meeting. Instead, its purpose is to "elicit feedback and foster collaboration," which is why the increment should be presented as a demo (and hence why it must be "potentially releasable" and "working," see Section 4.3), so that the review can be hands-on, which enables creative thinking and valuable insights for improvements.

On the other hand, this also means that the development team can only demonstrate work that is entirely "done" (see Section 4.3). However, unfinished work should become transparent, and it should be discussed what problems the development team ran into during the sprint and how they were tackled [237].

While a demonstration helps gather feedback, Rubin makes clear that it is not the first and foremost aim of the sprint review, which he describes as "in-depth conversation and collaboration among the participants to enable productive adaptations to surface and be exploited" [226, p. 370]. In that sense, a demonstration of the working increment is not the primary concern of the sprint review. However, it represents

14 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 369]

a *focal point* for the conversation and provides something concrete all participants can inspect and elaborate on.

Nonetheless, it is not only the increment that gets reviewed but also the overall context of the development. In order to decide how to proceed with development during the next sprint and what steps would be most valuable for the product, it is, for instance, important to know whether the market condition or potential use of the product has changed in the meantime. In addition, invited customers might also want to examine certain aspects of the project management, like time or budget constraints, as well as rough ideas of functionality and capabilities of the product for future releases.

Overall, the outcome of the sprint review is a set of identified features, enhancements, and future ideas, all of which are incorporated into a revised product backlog, which then constitutes the basis for planning activities of the next sprint.

4.7.4 *Sprint Retrospective*

While the sprint review concerns the product, the *sprint retrospective* is about inspecting the development *process* (see Figure 4.16).

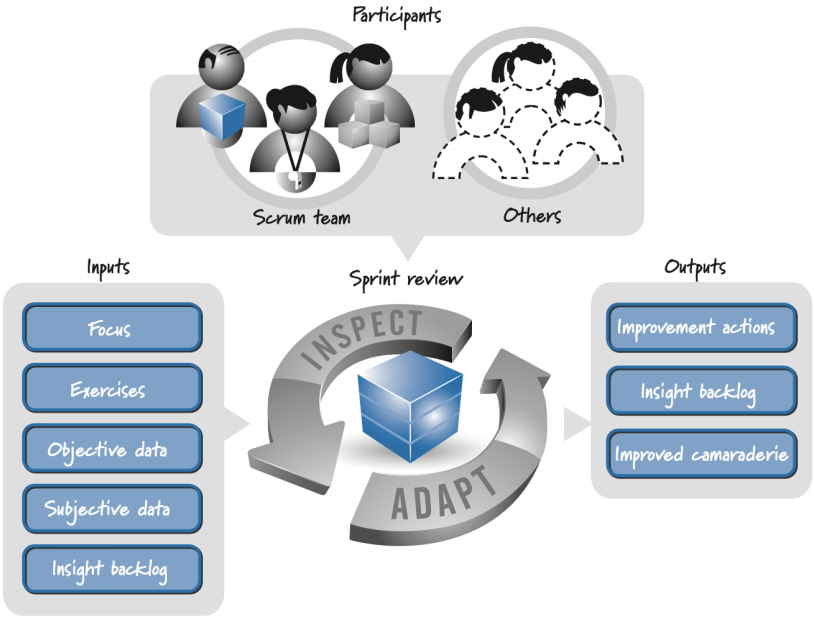


Figure 4.16: Sprint retrospective[15]

It is held after the sprint review and before the next sprint planning so that it represents the concluding meeting of the sprint before the

15 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 381]

next cycle begins. During the meeting, the inspection of the process is, for example, concerned with aspects of interpersonal communication, relationships between people, or issues of the environment, like the tools used for development. Anything affecting the product's creation during the sprint could be discussed.

Due to the broad range of topics and because everybody was involved in the development process, the sprint retrospective is of natural importance to any team member, and therefore, it is mandatory to participate. Facilitating the identification of relevant topics should be the particular concern of the Scrum master because, as the accountable person for the Scrum process, he or she knows best what worked well and what kind of problems existed during the sprint. With ongoing discussions, the Scrum master should also ensure that the atmosphere of the meeting stays positive and productive by encouraging the team's unstinting willingness to improve its Scrum implementation and the way of working.

Insights on how to improve the process are collected in an improvement backlog

Thus, the overall goal of the sprint retrospective is not only to create a shared awareness about concerns of the development process but, more specifically, to generate a *plan* for enhancing the Scrum process represented by an ordered list of so-called *insights* or *improvements*, which are concrete ideas about things to change for a better work process [226, p. 376].

Each sprint must include at least one improvement

From this list, at least one improvement must be considered during the next sprint planning, which then becomes part of the sprint backlog. Similar to the ordinary sprint backlog items, the insight is also broken down into a series of actionable steps so that its achievement can be verified during the next sprint retrospective. This makes the sprint retrospective the "sprint review" for previously identified insights. In contrast, the implementation of identified improvements in the next sprint is the adaptation to the inspection of the Scrum team itself, so that, in conclusion, the Scrum team does not only improve on the product but also on the process of building it, which according to Rubin can significantly affect the overall quality of the outcome [226, p. 376].

While the Scrum Guide emphasizes that the whole Scrum team is attending the sprint retrospective, some experts like Lacey point out that the main focus is on the members of the development team and their self-organized collaborative way of working since they are responsible for building the product. Therefore, it is in the interest of the development team that the product owner should not always be present in the retrospective, for instance, if developers would feel inhibited from speaking freely about their problems [152, p. 194].

On the contrary, Rubin picks up on that point and makes clear that "the product owner is a critical part of the Scrum process and as such

should be part of discussions about that process" [226, p. 377]. He argues that excluding the product owner from the retrospective to ensure honest discussions between the participants strongly indicates a lack of trust between the product owner and the development team. This lack of trust, however, may lead to severe problems and hence should be targeted by the Scrum master immediately by individual coaching to establish a safer and more trustful environment. In addition, he remarks that the participation of the product owner is essential since he is "the channel or conduit through which requirements flow" and, therefore, "critical to achieving the fast, flexible flow of business value" to the team [226, p. 377]. For that reason, the product owner's activities should also be addressed during the retrospective, for instance, whether product backlog items are well-groomed and clearly specified by the start of the sprint planning event.

Apart from the Scrum team itself, other persons like stakeholders or managers may also join the sprint retrospective by invitation, which is neither mentioned nor forbidden by the Scrum Guide. However, there is consensus among experts that although transparency is one of the framework's core values, companies have usually not (yet) established a level of agility and safety to promote the regular attendance of non-Scrum team members [226, p. 377].

The sprint retrospective is timeboxed to three quarters the hours of the weeks of a sprint

Regarding its temporal duration, as with all other meetings, the sprint retrospective is time-boxed and limited to *three hours for a one-month sprint*. However, for individual teams, the meeting length heavily depends on several factors. First of all, shorter sprints are tantamount to more frequent process evaluations, resulting in a greater likelihood of well-established procedures, whereas longer sprints usually need more extended reflection because they are more likely for process errors to creep in and have a higher chance that issues like bad habits are manifested.

Second, the length of the sprint retrospective depends on the team size. Smaller teams are easier to handle in terms of the process, so they usually need less time to inspect how they work. Several other factors, such as the team's experience, may also affect how long the team needs to identify improvement needs. For this reason, Derby and Larsen noted that good sprint retrospectives can also be substantially shorter. They may, for instance, only take 15 minutes to have a significant impact on the overall process quality [73, p. 17].

4.8 WHAT SCRUM LEFT OUT: DE FACTO STANDARDS

So far, all backlogs have been introduced without even describing what an individual item looks like. That is because the Scrum Guide leaves this design decision entirely to the Scrum team. However, Scrum

does regulate that elements of the product backlog must contain a *description*, *order*, *estimation*, and *value* but does not deliver examples or techniques for practical application.

For instance, Scrum is not telling *how* to specify a product backlog item. Is it textual or by using some formal specification language like UML? Nor is any information given about how the work required to implement a feature could be estimated so that the development team can confidently claim to deliver that particular feature within the limited timebox of one sprint. Is the development team, for instance, anticipating working hours, or how does estimation work?

These questions are meant to illustrate that Scrum has to be considered a *framework* rather than a fully formulated methodology, as explained in Section 4.1. It is particularly mentioned here since there are existing de facto standards for the specification of requirements and their estimation. However, what belongs to the core of Scrum and what does not should be very clear. That is because, as will be shown in later chapters, it is Scrum's dichotomy - being very strict in terms of its rules on the one hand and being very open to variances in practical application on the other hand - that may lead to severe problems with its implementation, especially when people begin to introduce variances to the essential rules as well.

That being said, the following sections will introduce techniques, concepts, and artifacts outside the official Scrum framework that became de facto standards and, therefore, can be found in nearly all Scrum-related development projects.

4.8.1 *User Stories*

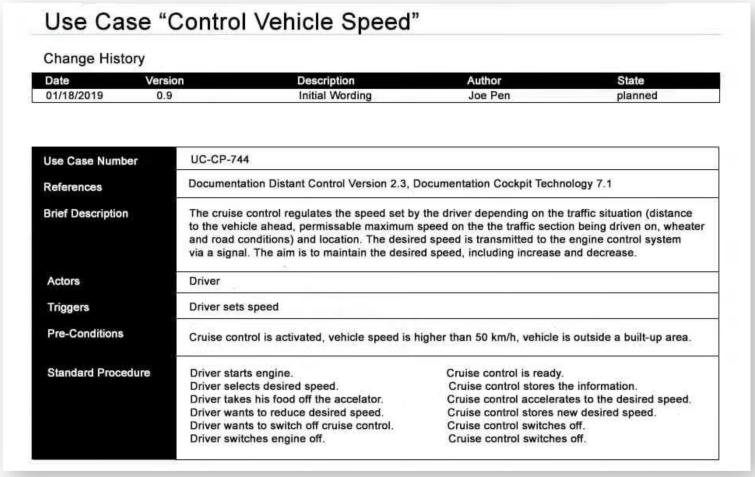
Originally, user stories stem from Extreme Programming and were introduced in the late 1990s as part of the "planning game" to define the scope of a project (see Page 65) [17].

Use case vs. user story

In contrast to *use cases*, which were introduced by Ivar Jacobson in 1987 as "a special sequence of transactions, performed by a user and a system in a dialogue" [301], and which were usually realized by formal specifications and diagrams (see Figure 4.17), user stories represent a *narrative description of a use case that fits on one index card*, as shown in Figure 4.18 [16].

Very soon, they were adopted by other agile approaches, and it became apparent that a user story differs from a use case. For instance, in 1999, Imaz and Benyon only treated them as the "first artifacts" used to describe interactions but stated that something formal, such as use cases, is needed for implementation purposes [121].

Three years later, Rachel Davies contributed to the discussion "user story vs. use case" by saying that both have the common purpose of describing functional requirements, so the difference between these methods could appear to be just a matter of the level of detail or precision in their descriptions. In that sense, user stories could be seen as lightweight versions of a use case scenario, with fewer words and less formal constraints [70].



Use Case "Control Vehicle Speed"

Change History

Date	Version	Description	Author	State
01/18/2019	0.9	Initial Wording	Joe Pen	planned

Use Case Number	UC-CP-744
References	Documentation Distant Control Version 2.3, Documentation Cockpit Technology 7.1
Brief Description	The cruise control regulates the speed set by the driver depending on the traffic situation (distance to the vehicle ahead, permissable maximum speed on the the traffic section being driven on, wheater and road conditions) and location. The desired speed is transmitted to the engine control system via a signal. The aim is to maintain the desired speed, including increase and decrease.
Actors	Driver
Triggers	Driver sets speed
Pre-Conditions	Cruise control is activated, vehicle speed is higher than 50 km/h, vehicle is outside a built-up area.
Standard Procedure	Driver starts engine. Cruise control is ready. Driver selects desired speed. Cruise control stores the information. Driver takes his food off the accelator. Cruise control accelerates to the desired speed. Driver wants to reduce desired speed. Cruise control stores new desired speed. Driver wants to switch off cruise control. Cruise control switches off. Driver switches engine off. Cruise control switches off.

Figure 4.17: Use case[16]

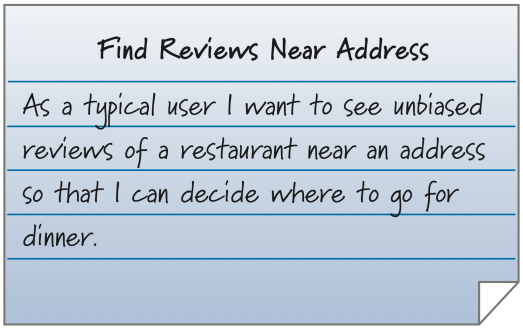


Figure 4.18: User story[17]

However, Davies has argued that the key differentiating factor is not the level of detail but rather that user stories are limited in scope because of the timeboxed aspect, which makes the activity of writing them "a powerful driver in the planning of software development iterations." Since then, user stories gained increasing popularity beyond Extreme Programming and, through various publications, developed into today's powerful yet simple format for specifying features and requirements within any agile development processes [158, p. 100].

16 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 83]
17 Source: "What is an Use Case?" by t2informatik [301]

4.8.1.1 *The Role-Feature-Benefit Template*

The Role-Feature-Benefit template is today's standard user story format and was invented in 2001 at Connextra[18], a former web development company in the UK, where the aforementioned Rachel Davies worked as a software developer in one of the earliest teams adopting the Extreme Programming approach.

At Connextra, early stories have just been feature requests written by people from sales and marketing, so developers struggled to have conversations because of the missing context and intention of the story [206]. As a result, they came up with the following template in order to remind people to pay attention to specifying not only "what" the feature is supposed to include but also "who" is the target audience and "why" the feature is relevant to them [3]:

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

The Role-Feature-Benefit template helps to establish a more user-centric development

According to Leffingwell, this template is exceedingly helpful because it spans the solution space or activity, explaining the product functionality and the problem space, including the delivered business value. Therefore, clearly expressing the benefit of functionality for a particular user would automatically lead to valuable product backlog items. Furthermore, this template would foster a user-centric approach to requirements engineering, which helps to empathize with real users, their needs, and problems [158, p. 103].

However, a blog post by Antony Marcano explains that this template is often misunderstood in the sense that user stories are analogous to product features or just a different way to write old-style software requirements [165]. He argues that people still capture product features and business benefits but only dress them in the template above. That is why he exhorts to think of a user story as "a short story that *a user will be able to tell* about what they want to do and why they want to do it." This thinking represents a shift in perspectives, where the focus is not on a particular feature of the product but rather on what the user will be able to *achieve*. As a result and in contrast to the traditional way of specifying requirements, a single user story may lead to changes of *multiple features*.

According to Marcano, it is important to understand that user stories and the Connextra template were actually introduced as a new form of specification, intended to invoke a paradigm shift that should help to step out of the "feature-ish" way of thinking and shift the focus on *user needs*. In that context, he quotes Jeff Patton, who states:

18 Hence, the Role-Feature-Benefit template is also known as the "Connextra format."

"Stories aren't a different way to write requirements, they're a different way to work." — Jeff Patton [165]

However, on its own, the Role-Feature-Benefit template was not enough to strengthen awareness about the paradigm shift due to the problems mentioned by Marcano. Hence, further contributions were made, leading to today's fundamental principles about how to gain the most benefit when specifying user stories correctly. Three of these contributions are presented in the following sections.

4.8.1.2 *Critical Aspects of User Stories: the three C's*

In 2001, Ron Jeffries pointed out that user stories have three critical components, nowadays known as the "three C's" [130]:

- *Cards* (their physical medium),
- *Conversation* (the discussion surrounding them),
- *Confirmation* (tests that verify them).

User stories should be written on cards

First of all, user stories must be written on *cards* because, with this, a requirement cannot be described to the last detail. The card's size limitation naturally narrows the description down to just enough information to act as a reminder of what the story is about. Moreover, a card represents the bundled, concrete, and atomic representation of the yet non-existent and, therefore, abstract requirement. Jeffries states that "the card is a token representing the requirement." What he means is that cards are manageable, and people can *interact* with them. During planning activities, notes and cost or priority labels can be added, and cards can be passed around to developers for implementing the story and back to the customer once the story is complete and ready to be reviewed.

User stories foster ongoing conversations

The restriction in specification also makes a card the medium of *conversation*, which is Jeffries's second aspect and means that exact details of a user story have to be communicated via verbal conversations. Just as the elements of the product backlog are highly dynamic, so is the conversation about a user story, not a singular event but an ongoing process. Short and bidirectional feedback loops of verbal communication help to avoid misinterpretations and foster a shared understanding of the envisioned functionality [226, p. 82]. Thereby, a card is a placeholder during planning activities, which acts as a constant reminder to hold conversations about details of the user story whenever needed.

Finally, this leads to the before mentioned paradigm shift of user stories - *from writing* about features *to talking* about them [57, p. 238]. These conversations are a key tool for Scrum teams because they

"enable a richer form of exchanging information and collaborating," which ensures that requirements are discussed and finally understood by everyone [226, p. 84]. This, in turn, is also reflected by one of the twelve principles behind the Agile Manifesto (see Chapter 3.6.4), suggesting that "the most efficient and effective method of conveying information to and within a development team is face-to-face conversation" [86].

User stories should include confirmations

In terms of the Scrum roles, a user story serves as a two-way promise between the product owner and the development team members. According to Cohn, developers promise to talk to the product owner before beginning to work on the story, which frees the product owner from concerns that every last detail must have been written on the card. Likewise, the product owner promises to be available when developers need clarification on details so that the development team can trustfully commit to the goal and workload of a sprint without knowing all of the minutiae upfront [57, p. 239].

However, Jeffries did not limit conversations to just verbal communication. Instead, he suggested supplementing discussions about user stories with other documents to provide more clarity. These might be UI sketches [226, p. 84] or, at best, executable examples, which he called *confirmations*.

Acceptance criteria define how a story can be verified

While the card represents the very essence of the user story, and its details are elaborated during conversations, Jeffries stated that a great level of uncertainty about the outcome is still likely. However, there should be no attempt to compensate for this lack of clarity with formal prerequisites, such as use case definitions, more sophisticated UI sketches, or extensive documentation. Instead, he proposed that conversations should lead to clear and testable *conditions of satisfaction*, also known as *acceptance criteria*, which are added to the card and represent a mutual agreement (in terms of Scrum between product owner and development team) on how the correct implementation of a story should be verified (see Figure 4.19).

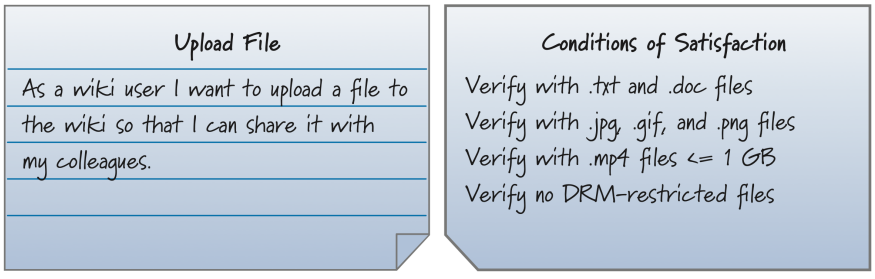


Figure 4.19: User story with acceptance criteria[19]

19 Source: "Essential Scrum" by Kenneth S. Rubin [226, p. 85]

According to Jeffries, this confirmation provided by the testable acceptance criteria makes the simple approach of card and conversation practically feasible and allows to settle the details of what needs to be done into clear expectations [130].

For the development team, these conditions determine the scope of work and define when the story is complete. Ideally, they are transformed into automated tests so that passing these tests confirms the correct implementation of the user story[20].

In any case, when developers demonstrate that the working increment at the end of the iteration complies with the specified criteria, expectations are confirmed, and so trust is built between the product owner as the representative of the customer and the development team. It is, therefore, all the more important that the development team has a clear understanding of the conditions of satisfaction, especially since elements of the product backlog are emergent and detailed appropriately (see Section 4.5), which means that acceptance criteria are also subject to change as long as the story has not been selected for a sprint. That is also why Rubin added "clear and testable acceptance criteria" to his exemplary definition of ready (see Page 93) as a prerequisite for the sprint planning meeting.

4.8.1.3 *The INVEST checklist*

The INVEST checklist helps to assess a user story's quality

While Jeffries's "three C's" describe the core philosophy of user stories and the relationships between cards, conversations, and confirmation, the *INVEST* checklist helps to assess a user story's overall *quality*. It originates in a web article by Bill Wake from 2003 [276] and was brought to a broader audience in 2004 by Cohn's book "User Stories Applied" [55]. According to Wake, the acronym INVEST serves as a reminder for characteristics qualifying a user story as "good."

***I**ndependent*

At first, stories are meant to be *independent* because this allows scheduling and implementing them in any order. This would be way more difficult if stories were interconnected by the common types of dependency identified by Wake, which are *overlap*, *order*, and *containment* [277].

Overlapping stories in terms of functionality may lead to severe problems because of confusion about whether sub-functionality is covered at all or more than once due to the functional overlap.

By contrast, *order dependencies* (user story A must be implemented before user story B) may complicate the plan to deliver the most valuable stories first (if B is more valuable than A), but Wake mentions

20 This approach is known as "test-driven-development."

that this type of dependency is usually harmless, since "the business will tend to schedule these stories in a way that reflects it."

The *containment dependency* results from a hierarchical organization of stories (user story B is contained in user story A) by using superordinate elements, such as "epics" or "themes," which bundle stories under a certain aspect. Wake argues that this hierarchical structure encourages a "depth-first" strategy for scheduling the implementation epic by epic, whereas a schedule driven by value usually contains stories covering aspects of the whole system.

Negotiable

A good story is also *negotiable* and not an exact contractual obligation. It captures the essence but leaves room for the customer and developer to work on details during the development collaboratively. This aspect also hints at the evolutionary design of user stories as a response to change from customer feedback and reflects the third key value of agile software development, defined by the Agile Manifesto as "customer collaboration over contract negotiation."

Valuable

Furthermore, stories must be *valuable* first and foremost to the customer. This has been addressed before in the rules of Scrum's product backlog, where the value of an item is reflected by its ordering. Wake targets this characteristic more technically by saying that value is a matter of breaking functionality down into a set of stories, where each story is a vertical slice through different implementation layers. This means the functionality of a story is a complete bundle and may affect all layers, from data storage (database layer) to the user interface (presentation layer). He mentions that developers tend to implement systems layer by layer. However, for instance, a complete database layer has little value to the customer if it cannot be used right from the beginning in its intended context.

Estimable

Moreover, good stories can be *estimated*. It must not be exact in terms of person-hours, but on a level, so that the implementation effort of stories can be compared against each other to help the customer (or in the case of Scrum, the product owner) to decide on a schedule. As shown in chapter 4.8.2.1, user stories are usually estimated with an abstract unit called *story points*.

Small

The ability to estimate directly depends on a user story's size. Bigger stories are more challenging to estimate because they include more uncertainty about the scope. Therefore, good stories tend to be *small*.

Testable

Finally, good stories are *testable*. This characteristic is a direct match to the "confirmation" as one of Jeffries's "three C's," which have been described in the previous section.

Overall, the INVEST checklist is a very simple way to determine the quality of user stories. It did not reinvent the wheel since many aspects have been covered before and can be found in the explanations

of Jeffries's "three C's" and, in terms of Scrum, also in the DEEP acronym for good product backlog management (see Chapter 4.5). However, it did unify all aspects of managing the requirements in agile environments under one memorable term. Therefore, it contributed a lot to the success of user stories as the de facto specification technique of software requirements in agile environments.

4.8.2 *Estimation Techniques*

As explained in Chapter 4.5, product backlog items must be estimated before being considered for the next sprint to plan the amount of work the development team can handle and balance the workload of an item against its proposed business value.

Usually, estimation occurs during the sprint planning event (see Chapter 4.7.1), in which the Scrum team agrees on a feature set for implementation during the next sprint. While the product owner presents the items that he considers most valuable and most important, he clarifies open questions and explains the acceptance criteria to the development team. After that, the development team has to assign an estimation value to each item representing the *size*, i.e., the estimated workload to transform the requirements into potentially releasable code. This estimation, in turn, serves as input to the product owner to balance business value against implementation effort, which may lead to the re-prioritization of items.

As an example, it could turn out that two items, which the product owner initially treated as indispensable for the next sprint, have been estimated by the development team with low effort for the first item and a very high amount of work for the second one. It is then up to the product owner to decide whether the more complex item should still be selected for the next sprint or if several other items (each with less effort but equal in the overall sum) may provide more value to the customer. It is also possible that the product owner might split the bigger and, hence, costlier item into several smaller ones so that important sub-parts could still be delivered with less effort.

Estimation must be quick to drive conversations and decision-making

However, what this example is meant to illustrate is that Scrum is such an agile and, therefore, dynamic process that it needs mechanisms and tools for making decisions *quickly* and *on the fly* since these decisions may immediately pose new questions and trigger new actions accordingly.

For that reason, estimating the workload of product backlog items should be just enough for the planning process to continue and hence be *abstracted* from the particular times it takes to implement its individual facets, like designing the user interface, coding the backend,

testing all functionality and integrating everything into the existing codebase. Cohn makes this very clear by saying that it is "one estimate, not many" [56, p. 46] that is assigned to a user story to prevent discussions about details that are not relevant for a rough estimation. This roughness of estimation is especially important since agile development methods generally encourage *change over following a plan* (according to the fourth value of the Agile Manifest, see Chapter 3.6.4) and Scrum, in particular with its time-boxed meetings, balances the effort and investment in planning with the knowledge that each plan will certainly be revised through the course of the project.

The following sections will introduce two different metrics and one particular technique for acquiring a quick joint decision and mutual agreement concerning the estimation value of a product backlog item.

4.8.2.1 *Story Points, T-Shirt Sizes and Velocity*

The first thing that comes to mind when estimating the size or workload of an item is the time it would take to implement its requirements, for example, by guessing the possibly performed person-hours or person-days. While this time-honored approach is still common for traditional software development projects, it is nowadays entirely discarded by the agile movement [259].

Estimation should not be based on time because of human's "planning fallacy"

This is because humans are not good at estimating hours or time for future tasks in general. Although research was aware of this phenomenon before, it gained more popularity when the term "planning fallacy" was first introduced by Daniel Kahneman and Amos Tversky in 1979, describing the cognitive quirk that humans tend to underestimate the amount of time a project will take and display an optimism bias when estimating time for their own future tasks [133]. Over the years, many psychological studies and research results from social sciences have confirmed their findings and investigated various causes for this effect [42]. Findings include temporal frames that influence the subjective distance to deadlines [229], missing awareness that most previous predictions were overly optimistic [43], sociological effects of group settings, in which people want to make a good impression and please others by "planning for success" [44] and hence tend to be more optimistic, while subconsciously insinuating being able to be faster, because of assuming to be better than others [50].

Estimation should be by relative measures

Agile development approaches have recognized the importance of this planning fallacy early on and started to *abstract the scope of work from its implementation time* within their planning activities. As a result, the implementation effort is not measured by the actual time for development but by comparing an item to other features and deriving a value that represents its *relative complexity*, which in theory could be in any unit of measurement.

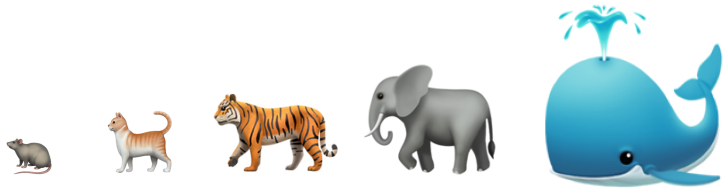


Figure 4.20: The "animal scale"

As an example, Figure 4.20 illustrates an "animal scale" so that features can be compared against each other and classified as being a "mouse," "house cat," "tiger," "elephant," or "blue whale." With this estimation scale, it becomes obvious that a feature estimated as a "tiger" is way more complex than a "mouse" feature. Similarly, a "tiger" feature is much smaller than a "blue whale" and will take less time to develop. But how much time *exactly*? This cannot be known in advance, so the agile philosophy promotes *relative measurement* due to social and psychological aspects of estimation, as mentioned by the Agile Alliance. According to them, emphasizing relative difficulty over absolute duration relieves tension between developers and managers when estimating workloads and development times because managers are likely to hold developers accountable for their estimations [4]. That is why agile teams need to understand that no value of estimation will ever directly relate to a particular time for development. Except that "bigger" features will usually take more time than "smaller" features and vice versa, but on its own, a "tiger" classification may stand for one day of work for team A, while it can also represent a whole week of work for team B.

During the years, there have been several proposed units within the agile community, ranging from the whimsical "gummi bears" unit, which was first mentioned by Ron Jeffries in 1999 [129], over the term "Nebulous Units of Time" (NUTs) coined by Joshua Kerievsky in 2003 [55, p. 87], to "T-Shirt sizes" and "story points," which are the two most commonly used scales today.

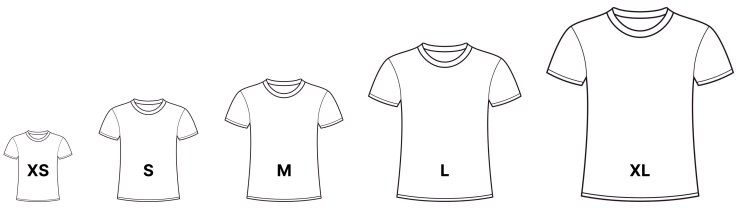


Figure 4.21: T-shirt sizes

As illustrated in Figure 4.21, an estimation using T-shirt sizes is analogous to the "animals scale" above. Both aim to simplify the comparability of the effort of two items using a simple-to-understand visual representation. In contrast, story points are just numerical values originating from when user stories became popular.

Figure 4.22: Story points in a linear scale

While in the beginning story points have been linear sets, as shown in Figure 4.22, it soon turned out that linear scales may provoke unnecessary discussions during planning activities [298]. Taking, for example, a scale of 1-10, then discussing whether an item should be estimated with 7 points instead of 8 points will not deliver any insights. It is not very meaningful and only adds complexity to the decision-making process. Instead, the essential insight is that the team estimates the item as something "big, but not the biggest imaginable," which means it is obviously not 10 points (the highest value from the scale above) but likewise not a medium value like 5 points.

Today, estimation is commonly based on story points

To fasten mutual decision-making, agile teams nowadays use non-linear scales, e.g., a modification of the Fibonacci scale (see Figure 4.23), which works well since it "reflect[s] the greater uncertainty associated with estimates for larger units of work" [56, p. 52].

Figure 4.23: Story points in a non-linear modified Fibonacci scale

Naturally, turning away from estimating the concrete implementation time of requirements has consequences for higher-level planning, for instance, when anticipating the timeframe for the next release of a particular feature set or planning the whole project's duration. In this context, it is important to remember that experiences from the software crisis (see Chapter 2.2.2) and the aforementioned "Chaos Study" from the Standish Group [105] have shown the full extent of estimation problems within traditional software projects, which finally led to the realization that it is nearly impossible for software projects to derive a release plan with a fixed scope (set of features), fixed budget and a fixed release date altogether [226, p. 311f]. Instead, at least one of these three parameters must be flexible to compensate for sudden and unexpected development changes that are known to occur very likely [57, p. 292f].

Fixed budget vs. fixed scope

For that reason, agile approaches began to introduce flexibility to project plans by *expressing budget as the possible number of iterations* and therefore by release schedules that either have a fixed scope but variable date and budget, so that it cannot be ruled out that more iterations are needed to implement the scope, or as an alternative, by schedules with a fixed date and budget but a variable scope, so that the number of iterations and therefore the costs are pre-determined, whereas the delivered set of features might deviate from the initial plan. For both approaches, however, it is necessary to know the team's performance, i.e., what it can usually deliver during an iteration.

The velocity represents the average team performance

This average team performance is known as *velocity*. In the case of story points as an estimation unit, it is easily computed as the average number of story points a team is able to complete during one iteration, using historical data of the last three sprints, for example. If the team uses non-numerical values for estimation, like T-shirt sizes, a velocity value can still be generated by mapping the abstract estimation scale to numerical values.

The Agile Alliance describes velocity as a "dimensionless" quantity [4], since for the reasons that estimation of items is abstracted from their concrete implementation times, velocity values of different teams cannot be compared. That means, on its own, there is nothing to gain from a comparison of team A having a velocity of 15 and team B having a velocity of 90. Besides having a lower velocity value, team A could nonetheless outperform team B. While comparing two teams' velocity values is meaningless, it is, however, an important metric for analyzing and comparing the performance of *one team over time* and helps to detect performance drops, e.g., as a result of impediments occurring in a sprint.

4.8.2.2 *Planning Poker*

Besides story points and T-shirt sizes as de-facto standards for estimation *units*, one particular estimation *technique* is widespread for quickly reaching a mutual agreement within agile group settings.

This consensus-based technique is called *planning poker* and was first defined by James Grenning in 2002 [98] as a variation of the *Wideband Delphi* method, which Barry Boehm and John Farquhar proposed during the 1970s [90] and which was brought to a greater audience by Boehm's book "Software Engineering Economics" in 1981 [28]. At that time, the Wideband Delphi method had already replaced existing forecasting approaches like the Delphi method, which relied on structured communication techniques, i.e., a panel of experts answering questionnaires over multiple rounds. In contrast, Wideband Delphi was characterized by greater participant interaction and communication.

Building upon this aspect and by adapting it to the typical agile planning setting, including user stories and story points, Grenning's planning poker gained immense popularity, especially since 2005 when it was mentioned by Mike Cohn in his famous book "Agile Estimation and Planning" [56], whose company later trademarked the term.

The planning poker game

The technique works as follows. At the beginning of a planning poker session, each developer holds a deck of cards consisting of all values of the estimation scale used by the team[21]. The product owner introduces a new backlog item, which is thoroughly discussed by the team so that everybody has a chance to ask questions and clarify the details. Afterward, each developer privately selects a card representing his or her estimate. Once all developers have made a decision, all selected cards are revealed simultaneously.

If it turns out that all participants selected the same value, it becomes the estimate, and the team moves on to the next feature. If no consensus is reached, the developers must discuss and share reasons for their individual selections. This discussion applies, in particular, to estimates at the outer ends because these outliers may reveal interesting aspects that the majority of the group may not have considered. Based on the agile philosophy, these discussions must be short and just enough to establish a sufficient understanding of each other's thought processes. With this gain in knowledge, the next poker round is triggered. Like before, every developer privately selects an estimate card until all cards are finally revealed simultaneously once everybody made a decision. This whole process is repeated until either consensus is reached or until it becomes evident that there is missing information, so the team cannot agree on a particular estimation value. In that case, the item was not properly specified in advance and thus has to be deferred to a later point in time [61].

Planning poker is especially powerful since it usually takes no more than three rounds for the team to decide on an estimate. With developers as being both the estimators and the ones that will implement the features, and through establishing a lively dialogue in which people are called upon by their teammates to justify their estimates, it is proven to enhance the accuracy of estimates and to eliminate underestimation as an effect of the previously mentioned planning fallacy [90]. For that reason, planning poker became today's de-facto standard technique within agile planning scenarios, including Scrum's activities

21 For story points, a deck usually consists of ten cards representing the values of the modified Fibonacci scale (see Figure 4.23). Some teams prefer adding the cards "0" and "?", with "0" expressing that there is zero effort to implement an item, e.g., because the feature was already implemented in a previous sprint, and "?" expressing that a developer is not confident in estimating, e.g., because of rising questions or missing information.

of sprint planning (see Chapter 4.7.1) or product backlog grooming (see Chapter 4.5).

4.8.3 *Tools for Monitoring Sprint Progress*

Chapter 4.6 introduced the sprint backlog more on a conceptual level as determined by the Scrum Guide, i.e., as a selection of items from the product backlog plus a plan for delivering the product increment and realizing the sprint goal. But what is meant by the "plan," and how do teams measure progress and whether they are on track to reach the sprint goal? Altogether, what does the sprint backlog look like in practice?

Indeed, the Scrum Guide does not provide any answers to these questions. That is because Scrum emphasizes the self-organization of the development team, which also includes complete freedom regarding the implementation of the sprint backlog itself. However, similar to how user stories became the quasi-standard of product backlog items, there are concrete visualizations of the sprint backlog and the progress of the development team, which became indispensable and are nowadays adopted by the vast majority of Scrum teams.

4.8.3.1 *Task Board*

A task board visualizes finished and remaining work

Recommended by many experts like Roman Pichler, Mike Cohn, Jeff Sutherland, and others, the *task board* is the most favored way of visualizing the sprint backlog and the plan for development [271]. According to Cohn, it serves a dual purpose: "giving a team a convenient mechanism for organizing their work and a way of seeing at a glance how much work is left" [56, p. 227].

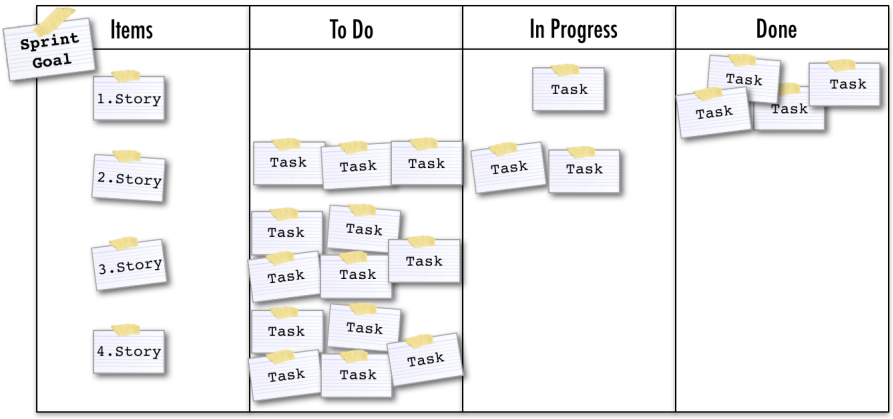


Figure 4.24: Task board[22]

22 Source: "Creating effective sprint goals" by Roman Pichler [210]

As illustrated in Figure 4.24, a task board is oriented in rows and columns, with each row consisting of one particular user story and several associated tasks. Tasks are organized in columns representing different states that follow one another. Since the development team is self-organizing, it is solely responsible for the task board. That means only developers define and create the tasks needed to deliver the feature as specified by the user story, which usually begins right after the whole Scrum team commits to the set of sprint backlog items as an outcome of the sprint planning event (see Chapter 4.7.1).

However, not all tasks of each story are specified at the beginning of a sprint. Usually, developers only create and assign themselves to tasks of stories they are working on or will work on soon. Over time, a task moves within the board from the left "To Do" column to the right "Done" column, symbolizing that this task is finished. In between, there should be at least one additional column for tasks that are currently processed by the development team, but, commonly, teams add more columns to the board, like, for instance, "Tested" or "Code Reviewed," which are often used to provide more details and a finer granularity about the different stages of implementation work.

Ideally, the task board not only reflects the state of tasks but also provides an overview in terms of the team's definition of done (see Section 4.3). This could be achieved by adding special columns with checkmarks that refer to the user story of each row, which makes the task board an extremely powerful process visualization of the current progress and all the remaining sprint work [56, p. 229].

4.8.3.2 *Burndown Chart*

The second widely adopted tool for tracking the progress of an iteration is visualizing the remaining work as a *burndown chart*, as illustrated in Figure 4.25.

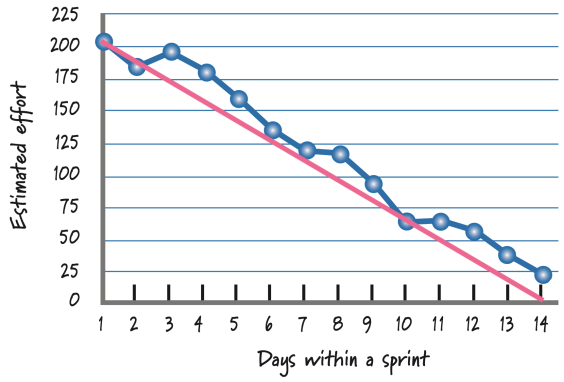


Figure 4.25: Burndown chart[23]

23 Modified version of source: "Essential Scrum" by Kenneth S. Rubin [226, p. 385]

A burndown chart visualizes the sum of remaining story points for each day of the sprint

While the x-axis shows the days within the current sprint, the y-axis represents the remaining workload as the sum of story points of all user stories that have not yet been implemented to their full extent [226, p. 358] at a given day. In addition to the real progress, the graph can show the ideal progress as a straight line from the initial value to zero on the last day of the sprint. Thus, comparing both lines makes it immediately apparent whether the team is on track to finish all stories by the end of the sprint.

Task boards and burndown charts are simple yet powerful tools for monitoring the progress of the sprint. In combination, they mutually complement each other since the task board uses the granularity of tasks to provide more details about the actual work status. In contrast, the burndown chart is on the level of user stories, which is the level of mutual understanding serving as a basis for all discussions between members of the Scrum team. In addition, the burndown chart promotes forecasting since it is much easier to determine trends within a graph-based visual representation than in a table-based representation.

Part II

PROBLEM ANALYSIS

The second part of this thesis constitutes the problem analysis. Although Scrum is known for being simple to learn, putting it into application is a stumbling process. But why?

What are the typical challenges for Scrum teams, and why do Scrum projects fail? Moreover, what role do existing Scrum tools play in managing the development process? Are they beneficial, or do they hinder the framework's correct implementation?

By providing answers to these questions, the objective of Part II is to derive a status quo of tool support and identify common challenges that prevent Scrum teams from reaching their full potential. Upon the outcome, the final part of this thesis will present a novel project management solution that addresses most of the identified problems.

5

RESEARCH QUESTIONS AND METHODS

Over the years since its introduction in 2001, Scrum has become by far the most commonly used agile process model for software development with a share of 87% out of other used agile practices in the industry [300].

Scrum is, moreover, known for reducing the time-to-market and boosting team productivity between 300% and 400% [20] when compared to more conventional project management techniques.

However, despite the continuous growth and adoption of Scrum in the industry, there is a fundamental downside to the agile development framework, which is revealed in the following statement of the inventors:

> *"Scrum is lightweight, simple to understand, difficult to master."*
>
> — The Scrum Guide [237]

5.1 RESEARCH QUESTIONS

The statement above raises the question: How shall something declared as lightweight and simple to understand result in something that is *difficult to master*?

Wouldn't we assume at first glance that true comprehension of any theory also encourages its practical application? So, what are the stumbling blocks causing the "simple to understand" Scrum theory to suddenly become complicated when put into practice, thus missing the chance to realize Scrum's full potential, leading to project prolongations and higher development costs? What kind of practical challenges exist for Scrum teams, and what are the consequences if people do not face them? These questions are aggregated into the first research question (RQ) of this thesis, asking:

What are typical challenges and issues for Scrum teams? (RQ 1)

Based on the answers given to this question, the overall goal is to identify potential starting points for a yet non-existing software solution for Scrum teams with the intention of finally inventing novel functionality specifically designed to address the previously identified problems.

For this reason, it is essential to analyze, as a first step, which kinds of tools Scrum teams use and what fundamental differences exist between them. Based on that, the investigation of the most commonly utilized tools and leading software products on the market should further complete the picture of which functionality exists for supporting Scrum teams in their development process and which aspects still need to be covered. Moreover, to understand how the existing functionality integrates into the special demands for agility and the particular needs of the Scrum team roles, it is also important to understand the user experience and usability of existing user interfaces and how they relate to the issues identified before. All of these aspects are part of the second research question, asking:

What is the status quo of Scrum tool support? (RQ 2)

In combination, both research questions should reveal an upper limit of Scrum tool capabilities, which are assumed to be related to the fact that existing tool support is fundamentally stuck in the still dominant interaction paradigm using graphical user interfaces consisting of windows, icons, menus, and pointers (so-called WIMP user interfaces). Claiming that the WIMP paradigm is not only unsuitable but an impediment to the new challenging demands of agile software development like cooperative, self-organizing teams heavily using face-to-face communication, the thesis will later propose that agile teams require a new kind of digital tool support.

Similar to the process of software development, which fulfilled the paradigm shift from a heavy, sequential, and document-driven process to a lightweight, incremental, and iterative way of creating software, agile teams might break the chains and begin incorporating tools relying on succeeding interaction paradigms. Regarding WIMP and graphical user interfaces, the succeeding paradigm is Natural User Interfaces, primarily based on multi-touch technology and interactive surfaces operated directly via finger input. Can this kind of technology add something new to the digital support of Scrum teams? This is the third research question, asking:

What could a novel Scrum tool look like utilizing NUI technologies for collaborative activities? (RQ 3)

All three research questions will be covered separately in the following chapters. Together, they form the problem analysis and represent the second part of this thesis.

Beforehand, the following section briefly overviews the different research methods used and how the questions were addressed over time.

5.2 RESEARCH METHOD OVERVIEW

Because of their diversity in subject matter, all three research questions have been treated individually in terms of the underlying research method. Moreover, an intertwined mixed-method approach was used so that each question was investigated by a combination of three research methods, as illustrated in Table 5.1. The combination of different methods is intended to target open issues and preliminary results from different perspectives, thus giving more substance to the topics and research outcomes.

RESEARCH QUESTION	METHODS
What are typical challenges and issues for Scrum Teams?	Literature review Ethnographic studies Interviews
What is the status quo of Scrum tool support?	Literature review Field studies Heuristic evaluation
What could a novel Scrum tool look like utilizing NUI technologies for collaborative activities?	Literature review Interviews Prototype development and evaluation

Table 5.1: Research methods

While more details about the individual methods will be given in the respective chapters of the research questions, this chapter aims to provide an overview of the different methods used and to illustrate their overall connection.

As seen in Figure 5.1, research for this thesis started in the middle of 2012. It lasted until 2019 and culminated in a startup funding project to transform the developed prototype of Chapter 9 into a commercial product. This dissertation was written simultaneously with the market launch and company-building process, which is why its finalization took until 2024.

However, despite the long research timeline, it must be clearly said that all of the analysis results presented throughout this thesis remain valid. This also includes the identified problems from investigating common Scrum software tools in Chapter 7, whose usability issues still exist in 2024 when this dissertation was finalized.

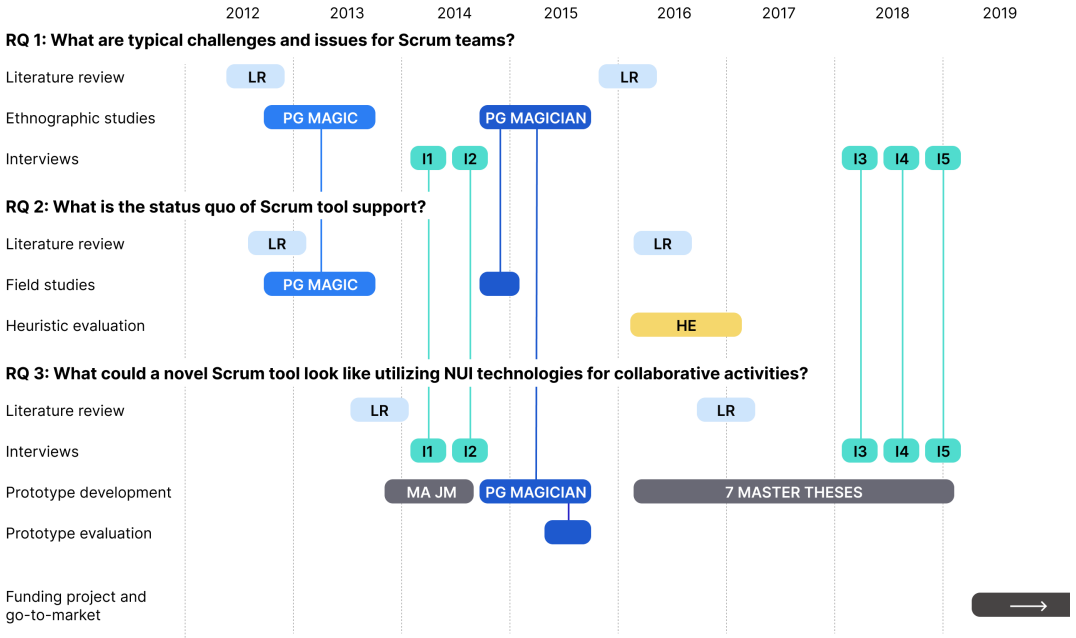


Figure 5.1: Research timeline

Behind this background, Figure 5.1 presents all research activities over the course of time. As can be seen, the individual research questions of this dissertation have not been investigated one after the other but with certain temporal overlaps. The following explanations help to understand how these activities relate to each other.

At the beginning of 2012, the first research question (RQ 1) was investigated by a literature review conducted prior to an ethnographic study about issues and challenges of a Scrum team represented by the student project group "PG MAGIC," whose members incorporated Scrum for their software development process. For managing the project, this group alternated between different Scrum tools, which therefore were field-tested in practical use, so that PG MAGIC not only contributed to investigating typical challenges and issues for Scrum teams but also delivered valuable data for the second research question (RQ 2) asking about status quo of Scrum tool support, which was also initially investigated by a literature review.

The purpose of PG MAGIC was to investigate novel interaction techniques and to identify potential application scenarios for future user interfaces based on the new interaction paradigm of "Natural User Interfaces" (NUIs), which do not rely on input via mouse or keyboard but for instance, via finger input using gestures and multi-touch technology. Near the end of PG MAGIC, the Scrum development process was identified as a promising application area for touch-based Natural User Interfaces. As a result, the third research question emerged, asking about the potential benefits of NUIs for managing the

Scrum development process, which was initially studied by reviewing existing literature.

Once it became clear that Natural User Interfaces had barely been researched in the context of Scrum and that the third research question (RQ 3) would address a nearly unexplored area, it became the main focus of attention. However, a holistic approach was chosen to take into consideration that RQ 1 and RQ 2 were still relevant since their results would provide the necessary backdrop against which the third research question should be investigated.

That is why all interviews conducted with Scrum experts and practitioners have been designed to simultaneously obtain data for RQ 1 and RQ 3. However, while the first two interviews in 2014 provided good results for RQ 1, respondents did not feel able to assess RQ 3 without being experts in the domain of HCI research and, in particular, without being able to experience a running system using NUI technology. As a result, another project group, "PG MAGICIAN," started in the last quarter of 2014 with the aim of developing such a running and testable system, building up on the first draft of a NUI-based Scrum project management tool envisioned by Julian Maicher, a former student of PG MAGIC, as part of his master thesis "Innovate Tool Support for Agile Scrum Teams" [164].

As shown in Figure 5.1, the contribution of PG MAGICIAN with a term of one year was manifold. First, and as already mentioned, the project group addressed RQ 3 and developed a usable prototype of a Scrum tool using NUI technologies and novel interaction concepts. Since the members of PG MAGICIAN incorporated Scrum, the team initially (within the first three months) organized their work by alternating between the same Scrum tools, which PG MAGIC has previously investigated. That way, research data for RQ 2 from the field study component could be gained from at least two different Scrum teams to check the initial results against possible side effects resulting from team composition. For the same reasons, the Scrum team of PG MAGICIAN was also the subject matter for the second ethnographic study of RQ 1. Lastly, the project group PG MAGICIAN self-evaluated the developed system during the last two-thirds of the term (for further details, see Chapter 9.4.1).

Between the end of 2015 and the beginning of 2017, each research question was re-assessed by further literature reviews to synchronize the data obtained through the project groups with results from scientific case studies of Scrum (in case of RQ 1), which have been published in the meantime, surveys about tool-support (in case of RQ 2) and papers about cooperative working scenarios using NUI technologies (in case of RQ 3).

In addition, a series of heuristic evaluations investigated RQ 2 and usability concerns of standard Scrum tools during 2016. They revealed significant gaps in functional and non-functional requirements, thus resulting in a poor user experience for Scrum team members within an agile work setting, as later explained in Chapter 9.4.2.

Subsequently and as part of RQ 3, these gaps have been analyzed by supervising various master theses concerned with particular Scrum aspects and enhancements of the developed prototype (see Table 5.2).

AUTHOR	THESIS
A. Gehle	Concept and prototypical implementation of digital support for sprint retrospectives in Scrum [91]
O. Blinova	Release Planning as long-term vision in Scrum [25]
S. Gerhardt	Supporting Decision-Making in Agile Development [93]
M. Stember	Conception and prototypical development of a task board as a tangible user interface to support the agile process model Scrum[257]
A. Quapp	Conception and further development of a tool to support the review process in Scrum [212]
M. Rose	Tool support for quality assurance in agile Scrum projects [222]
C. Klaussner	A Virtual Scrum Coach to Improve Agile Process Quality [140]

Table 5.2: Supervised master theses

The resulting system was further evaluated by more interviews in 2018 aimed at providing expert opinions about RQ 1 and RQ 3. In comparison to the first interviews in 2014, the respondents in 2018 were able to provide valuable insights about RQ 3 since they could experience the then-existing prototype and share their thoughts about the potential of NUI technologies for Scrum work settings. Afterward and because of the overall positive evaluation results, the developed prototype was transformed into a commercial product as part of a European startup funding program, during which the system was further evaluated by usability tests and expert interviews, as will be explained in Chapter 9.4.3.

While this overview should explain the interrelations between the different research questions and the methods used, the following chapters will provide more details about the individual methods and present the respective research results.

6

SCRUM ISSUES AND CHALLENGES

This chapter investigates the first research question, asking about the typical challenges and issues of Scrum teams. By laying the foundation for deriving possible software requirements for solving the identified issues, this chapter connects to the following one, in which the analysis to which degree existing tools support the identified problem areas will be one aspect of examining the status quo of Scrum tool support.

Beginning with details on the respective research methods, the findings of this investigation will be presented in an aggregated form by several individual subchapters representing different types of challenges.

6.1 RESEARCH METHOD DETAILS

As shown by Table 6.1, the investigation of Scrum issues and challenges was addressed by a combination of literature review, ethnographic studies, and interviews.

RESEARCH METHODS	SUBJECT MATTER
Literature review	Scrum books Scientific case studies Systematic literature reviews
Ethnographic studies	Project group MAGIC Project group MAGICIAN
Interviews	Scrum coaches Industry experts

Table 6.1: Research methods used for the analysis of Scrum challenges

The literature review is based on two primary sources. First, a selection of the highest-rated Scrum books recommended by industry experts, and second, several scientifically conducted case studies of Scrum implementations and published systematic paper reviews about Scrum problems.

In addition to the identified issues and challenges of Scrum teams taken from the literature, two types of qualitative studies were con-

ducted to augment the results with personal experience and observations. The first are two ethnographic studies of student groups using Scrum to manage their software development process, and the second are interviews conducted with Scrum coaches and industry experts to further augment the level of understanding of the problem domain by comparing personal experiences against third-party knowledge.

The following sections provide more details about each research method.

6.1.1 *Literature Review*

Initially, reviewing literature about common pitfalls of Scrum began in mid-2012, as illustrated in the research overview on Page 126. While this diagram shows two periods of time for simplifying reasons, it was, in fact, rather an ongoing and reoccurring process during which books and especially papers that have been published in the meantime were checked against new insights.

The decision to review both books *and* papers derives from the fact that the selected books are often written by authors with a background very close to the originators of Scrum, which means that the authors not only internalized its philosophy and principles but actively contributed to the development of Scrum from the very beginning. Hence, the book sections about Scrum issues and challenges have substantial value. On the other hand, these authors mostly describe Scrum problems in the form of narratives obtained from their vast experience in the field but do not provide scientific data. This, in turn, is the subject matter of scientifically published case studies about Scrum implementations or systematic paper reviews analyzing the set of paper publications against a particular research question. However, these authors may or may not be Scrum experts themselves, so interpretations and conclusions of obtained data must be treated with more caution compared to the old hands. Therefore, the investigation of both sources of literature should provide a more complete picture.

6.1.2 *Ethnographic Studies*

Both ethnographic studies were conducted over one year each with two student project groups using Scrum to manage their software development process. The first group, MAGIC, lasted from October 2012 to September 2013, while the second group, MAGICIAN, lasted from October 2014 to September 2015. Participants of both groups were master students (11 in the case of MAGIC and 12 for MAGICIAN) who

had to take part in a project group course due to the study schedule of the computer science master program at the Paderborn University.

6.1.2.1 *Study Context: Project Group MAGIC*

In order to provide contrast to the domains of software development that have been investigated through the previously mentioned literature review, the subject matter of the student project group was not as strict as it usually is in the industry. Instead, it was designed to explore the manifold research domain of human-computer interaction (HCI) by creating and investigating novel interaction techniques for different application scenarios, which were elaborated by the group members themselves. The openness for different aspects of HCI research is also reflected by the project group name, whereby "MAGIC" is an acronym for "**M**ultitouch **A**pplications and **G**eneral **I**nteraction **C**oncepts." Thus, the very nature of the project course was to promote creative thinking, and by developing something with personal belonging to the participants (nine of them male and two female), the overall rationale was also to quickly establish a conscious group thinking and team spirit, which was shown in Chapter 4.4.1 to be essential for agile work settings.

To illustrate the diversity of the projects and to point out that for all participants it was more than simply fulfilling the duty to take the project course[1], it is mentioned here that within one year, the group developed two tabletop games that strategically heavily rely on cooperative multitouch gestures for winning, a wedding planner application for tablets using gestures and optimization algorithms supporting simple seat place management, a digitally augmented location-based outdoor sports game for smartphones, and a tabletop application augmenting the daily Scrum meeting of developers by visualizing and providing access to data of the sprint backlog.

The latter is of particular importance because it was near the end of the project group when the participants realized that the technology and interaction techniques they have developed for different kinds of applications could also help enhance their own Scrum process. Given the freedom to experiment, the group quickly developed an initial prototype of a tabletop application for collaborative work planning of Scrum developers during their daily Scrum meeting (similar to a task board, as shown in Figure 4.24 on Page 118). It is worth noting that this marked the beginning of research associated with the third research question of this thesis, asking about the potential benefits of using novel interfaces for managing Scrum projects.

1 This fact could be seen from the given student feedback during the course evaluation.

6.1.2.2 *Study Context: Project Group MAGICIAN*

One year after PG MAGIC had ended, the second project group started to build up on the work of Julian Maicher, a former participant of PG MAGIC, who further developed the just mentioned tabletop application for the daily Scrum meeting in the course of his master thesis "Innovative Tool Support for Agile Scrum Teams" [164], in which he drafted a concept for richer support of Scrum teams by combining different kinds of in- and output devices.

As opposed to PG MAGIC, the problems to investigate were much more specific, again reflected by the project group's name, whereby MAGICIAN stands for **M**ultitouch **A**pplications and **G**reat **I**nteraction **C**oncepts **I**n **A**gile e**N**vironments. Hence, the purpose again was to investigate novel approaches and techniques of human-computer interaction, but this time, solely concentrating on the application scenario of agile team settings.

6.1.2.3 *Familiarizing the Project Group Participants With Scrum*

At the beginning of both project courses, the students had neither practical expertise with Scrum nor any other agile development methodology. However, they were all familiar with sequential process models and were used to longer planning phases before starting the implementation. That is because although the university's curriculum for the computer science bachelor course stipulated practical software development training in the form of the "Softwaretechnikpraktikum" - a one-year project during which a group of bachelor students mutually work on an implementation task - students were only familiarized with the traditional, document-driven development process.

In light of these circumstances, it was necessary for each group to provide a profound introduction to the essentials of Scrum. This was done by an initial seminar phase to establish theoretical knowledge about the underlying theory and various aspects of the framework.

In addition, a subsequent phase of three months was reserved for gaining practical Scrum experience in a sample "mini-project." During this period, the students worked on the task of creating an interactive tabletop game, but as a means of familiarizing themselves with all Scrum ceremonies and building the necessary team spirit. This introductory phase was vital for laying a strong foundation for their subsequent project work.

6.1.2.4 *Data Collection*

During the data collection process, the supervision of groups was combined with active participation in the development process. This involved taking on the role of the product owner and attending sprint planning, sprint review, and sprint retrospective meetings. While occupying the product owner role, it was important to pay close attention during these meetings. To ensure comprehensive coverage of important data points and to avoid disrupting the meeting flow, both note-taking and audio recordings were used. These recordings were analyzed immediately to identify team issues and challenges, transcribed, and then integrated with the meeting notes to create a detailed session record.

In addition to the mandatory meetings with regard to occupying the role of the product owner, participation also included a daily Scrum meeting once a week, silently taking notes on dialogues, interactions, and identified impediments as well as the Scrum issues observed, of which some have also been identified by the team members themselves during the regular inspection of the development process through the sprint retrospective.

While the students implemented Scrum using a sprint length of two weeks, there was hardly any difference to a professional Scrum team, except for one major. Due to the course specifications and the university program, the daily participation of all students could not be demanded, which conflicts with Scrum's daily meeting for the developers. However, to overcome this problem, members of both project groups organized their individual work and study schedules accordingly to agree on a compromise solution, defining three mutual presence days per week with a regular daily Scrum meeting. In order to also keep track of progress or impediments that occurred outside of the mutual working days (for instance, during weekends, when students decided to work on the project), the groups made use of a messenger application and decided that every participant had to publish answers to the three questions of the daily Scrum (see Page 99) once he/she worked on the project on a day other than the presence days.

These special organizational issues of a student Scrum team embedded in a university project course have also been considered for the analysis of the collected data. This means that particular problems resulting from the given circumstances have been excluded from the report since they cannot be applied to a general business setting.

6.1.3 *Interviews*

In addition to the ethnographic studies, the first research question was also addressed by interviews as another qualitative method to augment the knowledge level further and compare personal experiences against third parties.

6.1.3.1 *Participants*

In total, five interviews have been conducted, as shown in Table 6.2.

PARTICIPANT	BACKGROUND
Person 1	Group manager of a software development company building components and simulations for the automotive sector. Obtained PhD in computer science in 2003, so had theoretical knowledge about Scrum, but no practical experience. Took the role of the Scrum master in a project where Scrum failed.
Person 2	Self-employed certified Scrum coach and Scrum master. Worked at larger software development companies beforehand. Has several years of experience as a senior developer at a U.S. software company, which developed one of the Scrum tools presented in Section 7.1.3.
Person 3	CEO of a design and software agency using Scrum for managing the development process of web and mobile applications. Certified product owner with several years of practical experience.
Person 4	Self-employed venture capital investor and business angel. Beforehand entrepreneur with long-term experience as a senior software developer and CTO of different companies. Certified Scrum coach and Scrum master.
Person 5	CEO of a software development and agile consulting agency. Certified Scrum master.

Table 6.2: Interview participants

6.1.3.2 *Interview Design*

The interviews have been designed with a semi-structured approach following the interview guidelines of Turner [267] and pieces of advice for asking questions by Leech [157] and Dumay and Qu [79].

All interviews have been conducted on-site (except for the third, which was conducted via phone due to great distance) and at a time and place of the participant's choice to ensure that the person feels comfortable. At the beginning of each interview, an introduction was given to the participants, taking into account the preparation steps proposed by McNamara, consisting of explaining the purpose of the interview, addressing terms of confidentiality, explaining the format of the interview, indicating how long it might take and asking for permission to audio record the conversation for later analysis [176].

Each interview started with a couple of predetermined structured questions to gather information about the background of the participant and about how Scrum was introduced to the company. Afterward, the transition to a narrative flow was initiated by what Spradley calls a "typical grand tour question" [255, p. 86-88], asking, "Could you describe a typical sprint cycle of your Scrum team?"

Carefully and with particular attention to not disturbing the narrative flow of the conversation, the interview was guided by a previously prepared and tested set of briefly formulated questions ordered by themes. During the talk, the actual order and precise formulation of questions were adapted to the language and wording of the participant for gaining rapport [157] and thus allowing to evoke more profound responses from the interviewee [79].

Besides careful preparation, a particular challenge of the interviews was to identify Scrum issues and challenges that derive from a potential lack of understanding or appreciation of the framework's core elements and values since it was very likely that participants would not recognize their own weaknesses and hence would not be able to speak about them.

For this reason, the *soft laddering* method [106] was used in order to encourage interviewees to reflect on their motives while at the same time restricting the natural flow of the respondent's speech as little as possible. The motivation behind laddering is to "gain insight into the participant's underlying assumptions about the constructs in their web of associations" and "to elaborate on the meaning of his/her personal constructs by narratively forging links between them" [233]. This is achieved by repeatedly asking *how* and *why* questions, as shown in the following extract from the first interview.

question: What do you think, how close did you stick to the ceremonies and meetings as stipulated in the Scrum framework?

response: We have stuck to the rules Scrum prescribes and implemented all of the meetings. Regular planning, review - all of that.

question: How did you manage the retrospective?

response: It was not held on a regular basis, but depending on the needs and aspects of the sprint that I needed to communicate to the team.

question: Why was it not held on a regular basis, just like the other meetings?

response: If everything went well in the sprint, there was no need for the retrospective to take place.

question: Why do you think that there was no need for the retrospective in this case?

response: It would be a waste of time. If everything went well and I noticed that people managed to get their work done, there would be nothing to talk about.

question: How did you measure if everything went well during the sprint?

response: Depending on the amount of work that has been accomplished in the sprint.

In this case, the laddering technique gave first hints about the main motivation of the Scrum master to hold a retrospective, which, in his view, is to guarantee a certain level of throughput of implemented features. As it later turned out, he had fundamental problems establishing sprint retrospectives, where the Scrum team members should meet on equal footing and collectively elaborate on improvements to the process.

Accustomed to his position as group manager and due to the existing hierarchical structures of the company, he did not seem to question his own decisions, which became apparent because he did not realize that he, as the Scrum master, tried to introduce the framework without sticking to its fundamental agile values and continuously stressed that the project failed, because "the developers were incapable of implementing and adapting their manner of working."

6.2 RESULTS

For a better understanding, the results of investigating the first research question are initially presented in an overview to provide context between individual issues and challenges of Scrum teams that relate to each other. Afterward, individual sections for each finding will elaborate on the identified problems.

6.2.1 *Overview*

As explained in Section 6.1, the results of the first research question, which investigates challenges and particular issues of Scrum teams, have been derived from a combination of literature review, ethnographic studies, and interviews. Before providing an overview of the results, one particular paper from the literature is worth mentioning because it represents the closest match to comparable research in terms of the underlying question and presentation of the results.

In their paper "Exploring ScrumBut anti-patterns," [269] Eloranta et al. investigated "ways of potentially harmful mishandling of Scrum" in the industry and presented their results in the form of 14 anti-patterns, which represent deviations from core Scrum principles with negative consequences (see Figure 6.1).

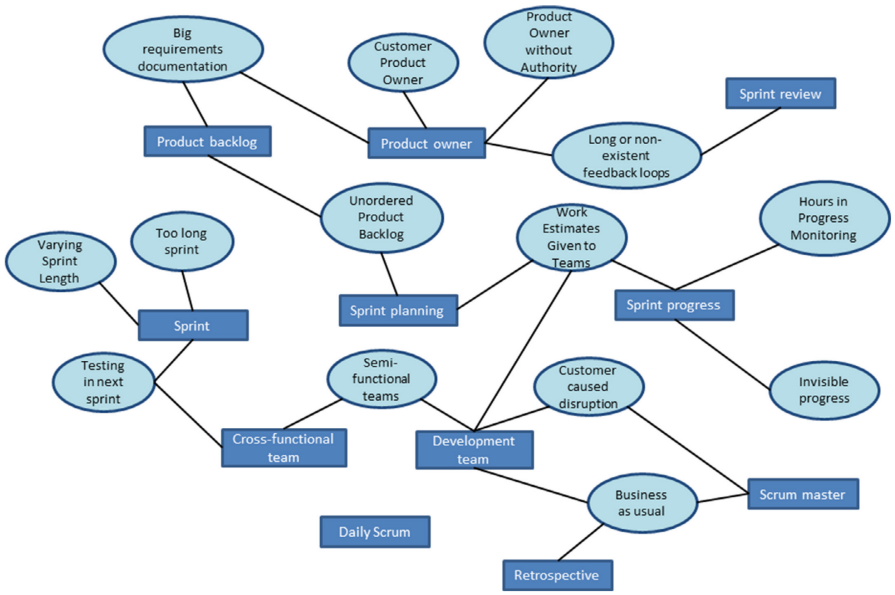


Figure 6.1: Scrum anti-patterns according to Eloranta et al.[2]

2 Source: "Exploring ScrumBut – An Empirical Study of Scrum Anti-Patterns" by Eloranta et al. [269]

However, two important aspects must be considered when looking at their results. At first, Eloranta et al. conducted 18 interviews but used a survey method with a fixed set of 48 predetermined questions [269], from which only less than half asked about details of Scrum. Moreover, on closer look, these questions did not manage to cover all aspects of Scrum (for instance, there are no questions addressing the daily Scrum meeting), and some of them are on a rather superficial level or formulated in a closed form (e.g., "Is the team self-organizing or who decides what will people work on?"). This closed form, however, possibly leads to rash answers from the interview participants so that deeper problems may not have a chance to be unveiled.

Second, the authors mostly asked for elements or specific aspects of Scrum but did not take a holistic view when interpreting the answers given. This means that some anti-patterns may result from outside influences that cannot be investigated by asking questions restricted to Scrum aspects only. For that reason, the interviews conducted for this thesis have been designed in a semi-structured form with a strong focus on open questions, which, in combination with the laddering technique, allow to shed light on the answers given from different perspectives for a more profound background analysis.

While Eloranta et al. identified 14 Scrum anti-patterns, the following results of investigating the first research question of this thesis revealed 42 Scrum issues clustered into eight challenge areas (see Figure 6.2).

6.2.2 *Challenge: Waterfall-Ish Environments*

Being embedded in a working environment that still uses the traditional software development process can be very problematic and will likely result in a Scrum team incapable of unleashing the full potential of being agile. That means, although teams embrace the agile idea and adopt basic Scrum principles like the roles, ceremonies, and planning activities, it is very common that the agile mindset is not lived within the company as a whole. Dave West explains that companies trying to adopt the agile approach often fall back on tradition because of long-term experiences with elaborated processes and management tools that are difficult to throw overboard. So, even if companies claim to be agile or adopted an agile development approach, the reality often is that *plans still drive the funding of projects*:

> *"The plan defines the project. It includes a detailed description of the tasks, resources, and time the project requires, translated into cost and time estimates. The project's justification is the difference between the benefits described in the business case and the initial estimates and plan."*
>
> — Dave West [287]

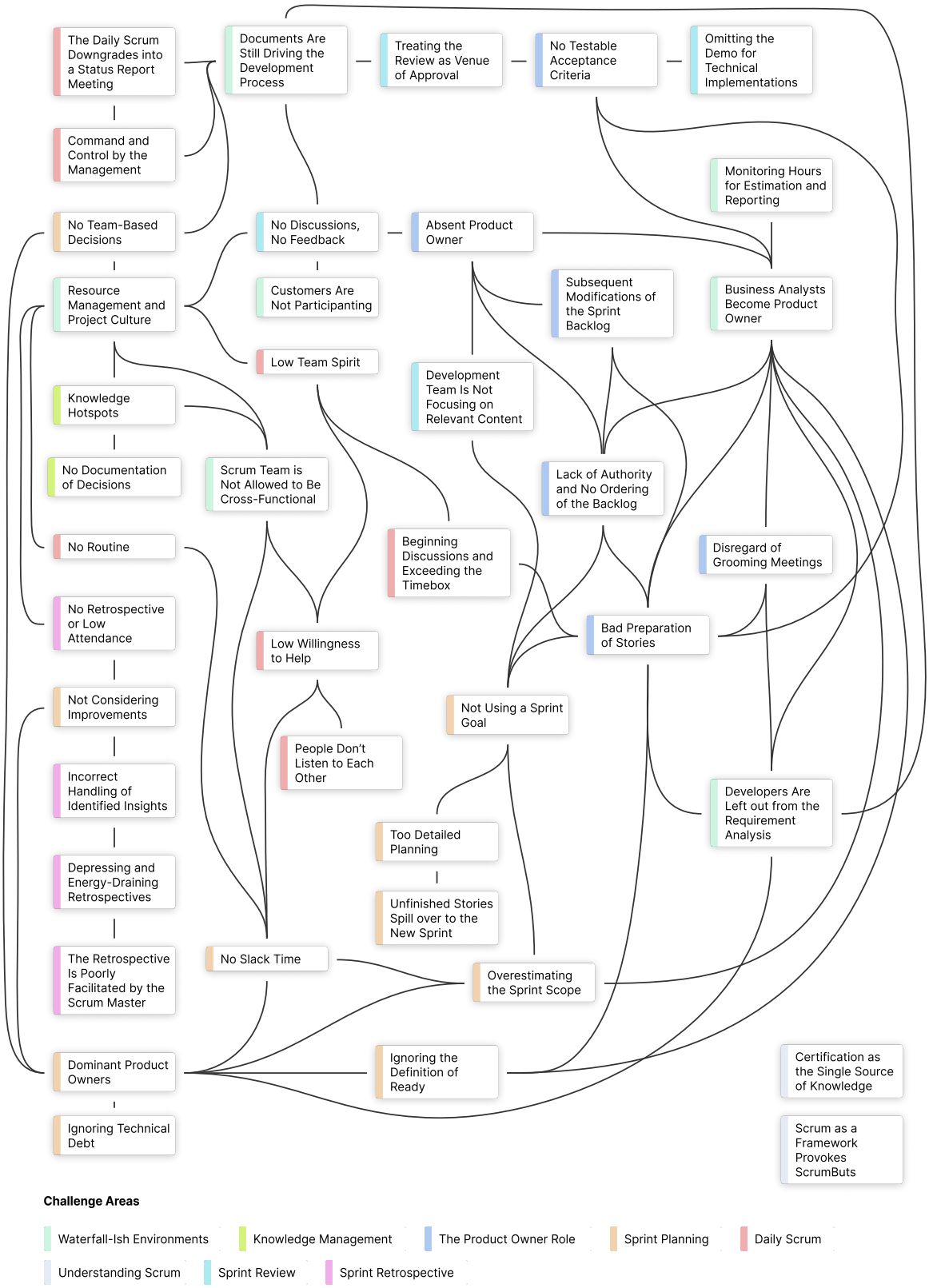


Figure 6.2: Scrum challenges and issues

If plans drive the funding of projects, a heavy amount of detailed specification is still shifted towards the very beginning of the development process, and overall, the waterfall model is not abandoned at all. Instead, it is either hidden behind a series of sprints still representing the sequential phases of the waterfall model, or Scrum is only incorporated within the actual "coding" phase. In both cases, claiming to be agile is, in reality, a process model that Dave West calls the "Water-Scrum-Fall" (see Figure 6.3).

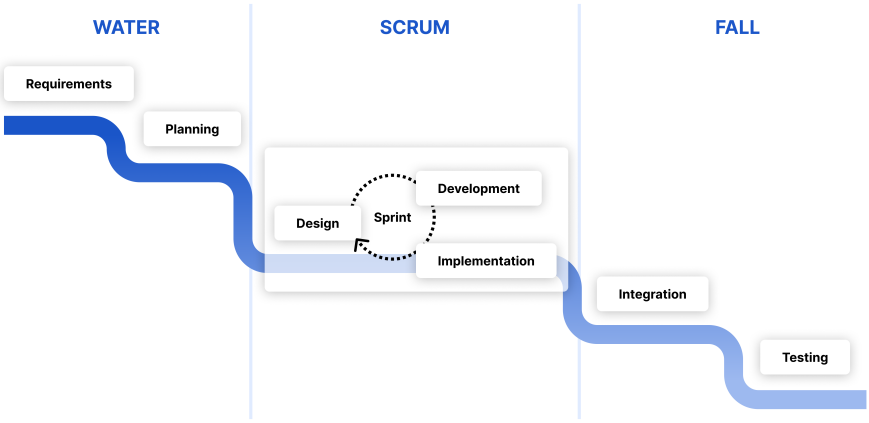


Figure 6.3: Water-Scrum-Fall

The problem with the Water-Scrum-Fall model is that agile teams are waiting for work due to discrete planning processes ("Water"). While they can accelerate during design, development, and implementation ("SCRUM"), the delivery is slowed by traditional and manual processes ("FALL"), resulting in slow feedback loops.

In principle, this process could work for a Scrum team that is independent of its non-agile environment. However, this is rarely the case. Instead, it is common for many interconnections and outside influences to exert significant pressure on the Scrum team, which then has to deal with traditionally established structures, like hierarchical management of the organization and business concerns. This causes the Scrum team to be unable to break the chains from traditional behavior, as illustrated by some of the following examples.

6.2.2.1 *Issue: Business Analysts Become Product Owner*

West explains that it is common for established companies to occupy the product owner role by previous business analysts or project managers, assuming that the associated competence areas are closely related. However, the relationship is not as close as it may look at first glance. While business analysts or project managers are competent in communicating the customer's intent to the development team, they often lack credibility and responsibility to really *own* the prod-

uct and drive the business by their own decisions [287]. In addition, they usually do not have enough technical knowledge to work closely with the development team. As a result, this misunderstanding of the product owner's role is causing severe problems like insufficient backlog management and slowing the team down in making decisions, potentially affecting the outcome and jeopardizing meeting the sprint goal.

6.2.2.2 *Issue: Resource Management and Project Culture*

Another aspect of grown management structures, limiting the potential for true agility, is the intent to maximize the utilization of human resources by *spreading individual developers across multiple projects* and spending their time on different problems in varying teams. However, the most central component of Scrum is a well-coordinated and experienced team that delivers software in close collaboration with its members. This also means that an established Scrum team should stay together across projects, not only to reduce the overhead in time for familiarizing with each other and socializing shared working practices but also to keep the knowledge that has been mutually gained. Therefore, re-composing teams for particular activities and slicing a person's time across multiple projects make it very hard for team members to organize themselves for close collaboration and hinder the opportunity to grow together. In addition, time slicing between projects includes considerable risk for a team to lose momentum because of the context switching. It, therefore, unnecessarily endangers the overall commitment of the team [287].

6.2.2.3 *Issue: Scrum Team is Not Allowed to Be Cross-Functional*

A further issue when making an agile development approach in a non-agile organization is that the development team is often not allowed to be cross-functional. Not in the sense that it is explicitly forbidden, but as a result of the Water-Scrum-Fall, as seen in Figure 6.3. Here, the separation of concerns as a tradition of the waterfall phases is very likely. A classic example of this is, according to West, testing. In Scrum, all testing activities are part of the sprint, ensuring the quality of the potentially releasable increment (see Chapter 4).

In contrast, non-agile environments often make use of *separate testing teams* and move the validation of functionalities and code outside of the development team, thus increasing time for correction of errors and leading to a loss of rapid feedback, which is essential for agile teams [287]. However, the problem relates not only to testing but, more generally, to any specialist departments, such as dedicated groups of architects, designers, etc., separated from the actual Scrum develop-

ment team. While these groups certainly have clear responsibilities for the separate stages of the Waterfall model, this is not the case in terms of Scrum, which, in contrast, is designed around a cross-functional team that is solely responsible for delivering the product increment, including any aspects of the software development lifecycle (design, code, testing, etc.).

6.2.2.4 *Issue: Developers Are Left out from the Requirement Analysis*

Similar to the previous issue, where the Scrum team lost control of specific aspects of development, it is also problematic when the team members, especially the developers, are left out of the requirements process. This is quite common in the Waterfall model, where business analysts define the requirements by creating documents describing the business problem, which then form the contract between business and development [287].

However, Scrum needs developers to be part of the requirements process since they are experts in technical feasibility and can provide valuable feedback. Therefore, the specification of requirements should not be the concern of business analysts separated from the Scrum team. Instead, close collaboration between the product owner and development team, in the form of grooming events and tight feedback loops, ensures that the team can cope with constant changes in the requirements, which are very common.

6.2.2.5 *Issue: Documents Are Still Driving the Development Process*

While traditional software development processes heavily depend on creating documents to mitigate the risk of the false implementation of requirements, the emphasis on documents has appreciable consequences for Scrum teams embedded in a waterfall-ish environment. In terms of the Water-Scrum-Fall, as illustrated in Figure 6.3, plans still drive funding of projects, which means that many (design) decisions are excluded from the Scrum team and anticipated by previous stages in the form of documents that stakeholders can sign off to start the actual coding. Not only is this detailed upfront specification against one of the core agile values ("Responding to change over following a plan," see Section 3.6.4), but it also affects the Scrum team itself in terms of the need to stick to the central values (see Section 4.4.1).

Commitment and motivation of the team members are at risk when predefined plans leave little room for errors and if developers do not understand the necessity for all of the Scrum ceremonies, which might be the case when many aspects of the product have already been signed off by specifications within documents. Therefore, the great

danger of this plan-driven development is to make the tight feedback-response cycle of Scrum, including its ceremonies for collaborative inspection and review of the outcome, superfluous.

6.2.2.6 *Issue: Customers Are Not Participating*

The conducted interviews revealed another critical issue caused by a traditional environment, which is out of scope even for a company implementing Scrum and the agile philosophy perfectly. That is a "traditional" customer who is unwilling or unable to contribute to the tight feedback-response cycle by regularly attending the sprint review and providing feedback to the Scrum team, although this is essential for refining requirements for future iterations.

This issue was also experienced and reported by Juyun Cho [52] as one of the significant problems of any agile development approach. While he links it to communication issues with the customer, responses in the interviews revealed that the reasons for this issue seem to be manifold and on different levels. For instance, the answers range from difficulties to agreeing on a regular date because of temporal overlaps between the customer and Scrum team schedules, over an insufficient understanding of their role as contributors of feedback, which enables iterative and incremental development in the first place, to direct refusal and lack of willingness to spend time and effort for something that should be the duty of others. The last reason was mentioned by one interviewee with long-term experience as a Scrum coach, and he referred to this as the "I already pay enough for it, so do it alone"-mentality of certain customers. Nevertheless, he also said that this kind of customer behavior is rather the exception than the norm and agreed with the common statement that it is primarily the lack of understanding of the agile model that causes low interest in actively participating in the development process.

6.2.2.7 *Issue: Monitoring Hours for Estimation and Reporting*

A Scrum team should always strive to deliver the most valuable product increment at the end of each iteration. Since during a sprint, the development team works towards this increment in an undisturbed and self-organized manner, the goal and the selection of items should be determined only once at the sprint planning event.

Here, it is crucial that the product owner has previously prioritized the backlog according to the items' business value for the customer. However, the decision about selecting an item for the sprint is not taken on the item's priority alone. Instead, it must be balanced against the implementation effort, which is estimated by the development

team, to optimize for the biggest return on investment. In this context, it is important to understand that this estimation serves two aspects. At first, it is a measure that allows the team to decide whether the item is, in fact, small enough so that it can be finished by the end of the sprint and, hence, becomes part of the increment. Second, the estimation is a *comparative* measure. For example, we can imagine two items with the same business value. When deciding which one to include in the sprint, it is sufficient to determine which item takes less effort, which may be achieved by comparing them side by side. This principle of comparative effort measurement allows quick decision-making because for prioritizing items according to the return on investment, it is not necessary to calculate the effort as an *exact* and absolute measure, e.g., in the form of working hours. In fact, discussion about exact working hours would slow down the planning process.

However, in hierarchical organizations, in which agile principles have not yet evolved to a mature level, project managers often use working hours as the calculation for feature or project prices [269]. Hence, this principle is also transferred to the sprint planning event when estimating items of the backlog. However, this may have severe consequences. At first, an estimate will always be inaccurate, especially at the beginning of the sprint. Even if an item is understood by the team and its scope is well defined by clear and testable acceptance criteria, there still is an amount of uncertainty. Hence, a discussion about whether an item takes, for instance, five or six hours to implement is useless and represents a waste of time. Second, estimating working hours also generates wrong expectations at the management level because of reports and forecasts of features to particular dates. Overall, this creates the illusion of being able to know everything in advance and carries the danger that the original estimates are clung to too much, and it is to be forgotten that they were, in fact, only estimates.

6.2.3 *Challenge: Knowledge Management*

As described in Chapter 3.6.4, agile methodologies introduced an alternative approach for managing knowledge, which is so fundamentally different from the traditional way that it is addressed by two of the four values of the agile manifesto:

"Working software over comprehensive documentation" expresses that the ultimate measure of progress is working software that the customer can review through live testing. This contrasts with written documentation that only describes what the software could do *in theory* but cannot provide any real user experience.

In addition, "responding to change over following a plan" indicates that all plans (and this includes a priori documentation) are subject to change. As a result of these two aspects, documents became less important as a criterion of customer approval, whereas the running software itself, which is constantly adapted to the customer's needs, became the first and foremost matter of success monitoring. Due to this shift, agile methodologies generally tend to dispense with external documents, significantly reducing the amount of overall documentation [52]. Instead, they claim the code itself should be the document since this naturally is the source of change that is inherently updated with any software alteration.

However, this gives rise to the following consequences.

6.2.3.1 *Issue: No Documentation of Decisions*

Given the dispense of external documents and developers placing more comments in the code, Cho identified that this leads to problems in terms of a missing global overview. Without any documents except the code itself, new developers joining the team at a later project stage had difficulties understanding why certain things were done in a particular way and, therefore, would appreciate specification documents for easy access to the system [52]. Furthermore, these difficulties apply not only to new developers but also to those working on parts of the system they have never worked on before.

Not understanding the decisions that led to the actual situation slows developers down. They will ask many questions to get clarification, which takes time away from implementing items of the sprint backlog, especially if these circumstances have not already been considered during the sprint planning event in terms of additional time for teaching and learning.

6.2.3.2 *Issue: Knowledge Hotspots*

Theoretically, the idea behind reducing documents is supported by Scrum's demand for self-organized and cross-functional development teams since this pursues sharing skills and mutual knowledge on the system, as explained in Chapter 4.4.3. That way, the absence of one developer can be compensated by others sharing the same knowledge so that the team's overall performance is not so heavily affected as to endanger the sprint goal.

However, the conducted interviews revealed that, in reality, development teams are often unable to cope with such situations because of *knowledge hotspots* of particular team members. This term describes knowledge about specific parts of the system that is exclusive to a

few developers or, in an extreme case, even restricted to only one developer. Knowledge hotspots make individual persons irreplaceable, and project success depends on those persons' availability.

This phenomenon was - with only one exception - mentioned by all interview participants regarding their own development teams and could also be clearly identified in both project groups. Interestingly, the respondents seemed to be conscious of the consequences of such knowledge hotspots in the case of a longer absence of particular persons. On further inquiry, they partially admitted to neglecting this issue for various reasons (mainly for reasons of time), whereas one respondent was very open and stated that his team members are closing their eyes to the consequences because they did not attach sufficient importance to the problem in the past and now feel like having reached a point of no return, where no one is confident to embrace change to the given situation, nor have any of the developers an incentive to work themselves into the "dark parts" of the system.

6.2.4 *Challenge: The Product Owner Role*

Compared to all of the other Scrum roles, the product owner role is probably the most challenging one, as shown by Eloranta et al. [269]. That is because this role is connected with a high level of responsibility since it is the central pivotal point through which various parties exchange expectations. In order to satisfy these expectations and to take ownership of the product, it requires versatile knowledge and a complex set of skills [15]. On the customer side, for example, the product owner has to understand and analyze the given problems before deciding how to deliver valuable features. For this, he or she keeps an eye on functional and non-functional aspects, which requires deep knowledge about the product itself, the domain and market trends, and the users of the future system.

Simultaneously, the product owner bridges the gap between the client and the development team and, therefore, needs the ability to communicate on equal footing with developers about various aspects of the implementation process. This applies in particular to the fact that the product owner also balances decisions regarding the further course of action upon technical dependencies, which must be considered when formulating the acceptance criteria of user stories, for example.

Overall, the product owner must be capable of acting on different levels, ranging from strategic to tactical to operational, when developing the product strategy and demonstrating it to all involved parties while continuously aiming at the best return on investment. Consequently, issues related to the product owner have a high risk of failing the whole project.

6.2.4.1 *Issue: Absent Product Owner*

During their studies about anti-patterns of Scrum, Eloranta et al. noticed that surprisingly often, teams reported not having a product owner at all [269]. In other cases, the role is assigned within a Scrum team, but the person in charge is not always available, for instance, because of part-time work or further responsibilities apart from the role itself [146].

This issue was also mentioned by three of the five interview participants, who stated that the absence of the product owner had caused several problems and unnecessary impediments in the past. In particular, it turned out that two product owners do not attend the sprint review on a regular basis, especially when the customer is not able to attend as well. In these cases, the team "self-inspects" the sprint result, and developers start to make decisions on their own. This, in turn, disconnects the development team from the customer feedback and is likely to lead to the delivery of incorrect or wrong features at the end of the project.

In addition, the respondents said that the lack of rapid feedback to arising questions of the development team has the effect that the team starts to maintain the backlog and determine the stories' acceptance criteria on their own, which is absolutely against the rules of Scrum because it softens the responsibility of product ownership. This is closely connected to the issue of lack of authority (see Section 6.2.4.2).

Overall, the interviews showed that the product owner's presence is crucial to establishing the communication channel between development and the client side. This role is in charge of answering questions *quickly* and providing valuable information on time so that availability never becomes the bottleneck of development progress.

In that respect, personal experiences as the product owner of the two project groups also showed that even the most careful preparation in advance for a temporary absence did not provide protection against this issue. In this regard, it must be noted that the product owner (me) was on parental leave for one month in each of the project groups. As preparation for each absence, a sufficiently large set of user stories was specified, which was prioritized and carefully discussed with the team before the parental leave so that each story fulfilled the definition of ready. The preparation ensured that everybody understood all of the acceptance criteria and that potential technical dependencies had been discussed. In addition, the groups agreed to change the sprint length from one week to two weeks. This change was made because of the offering to attend the Scrum meetings nonetheless to prevent further problems during the vacation.

However, neither project group made use of this offer. Instead, the development teams started to make their own decisions whenever questions or problems arose, which normally would have required an inquiry from the product owner. As a result, the outcomes showed significant deviations from the initially agreed acceptance criteria. When asked about this, it turned out that each development team had a member who felt encouraged to take the lead about the product and, driven by false ambition, treated inquiry as a form of weakness. This outcome was even more surprising since the relationships between the students and the product owner (myself) were intentionally built on trust and equal footing to promote an ideal agile cooperation.

6.2.4.2 *Issue: Lack of Authority and No Ordering of the Backlog*

Further issues identified by Eloranta et al. concerning the product owner role result from persons not being able to fulfill that role's duties. As a reason, they mention insufficient understanding of the broad remit and associated tasks, lack of motivation to use Scrum, missing interest in the product owner role, and fragmentation of responsibilities, especially in large organizations with multiple products and development teams [269]. Consequently, this would lead to the absence of authority regarding the product.

An example of missing authority might be that development team members are promoted as product owners when product managers or former business analysts are not interested in using Scrum or feel uncomfortable with agile thinking (see Section 6.2.2.1). Moreover, persons of a higher level of management might undermine a product owner's authority by interfering with decisions that should be taken by the product owner alone. Additionally, Eloranta et al. mention scenarios in which product owners did not have direct contact with the customer at all and, therefore, cannot provide information about the value of features to the customer.

In any case, the result of missing authority often is that the product owner cannot decide which items to implement and which to discard, making prioritization less meaningful. For that reason, many teams are observed to work with an *unordered* product backlog because of missing competence to do so.

Not only is this leading to a loss of vision about how to satisfy the customer needs, for instance, because of a team that is working on features that are of no value to the customer or will rarely be used, but it also promotes that the challenging features that are more difficult to implement are postponed, whereas features that are easy to deliver will be picked at first. This overall increases the risk for smaller problems to develop into severe impediments during later stages of the project.

6.2.4.3 *Issue: Bad Preparation of Stories*

The following three issues all fall into the category of bad backlog management prior to the sprint planning event and have been derived from the conducted interviews. The first is about a bad preparation of user stories, which could be identified in all of the respondent's answers to a greater or lesser extent.

At first, it turned out that many backlogs were maintained without clear knowledge about what aspects of a user story must be fulfilled in preparation for the sprint planning event. This goes hand in hand with not implementing a definition of ready (see Section 4.5). Second, some product owners do not use acceptance criteria at all or do not use them on a regular basis. Instead, they sometimes use continuous text to describe the expected result or, in many cases, do not provide any additional information other than just the story title itself, assuming that it is self-instructive to the development team.

However, as reported, insufficient backlog preparation can lead to severe problems regarding the subsequent sprint planning events. This is because it promotes discussions and leads to time-consuming queries when developers first have to understand the full extent of a story's description. More importantly, exhaustive sprint planning meetings have been mentioned as highly demotivating for the team and affecting the overall commitment to reaching the sprint goal.

6.2.4.4 *Issue: Disregard of Grooming Meetings*

Another issue affecting the overall quality of preparing the backlog before the sprint planning event is when product owners neglect to have grooming meetings with developers (see Section 4.5) and, therefore, treat every aspect of the backlog maintenance as their sole responsibility. While the latter is true according to the Scrum Guide, a complete disregard of grooming events nevertheless indicates a false understanding of Scrum's role model. It is often forgotten that grooming meetings are an essential part of the Scrum framework, too, and especially made to take feedback from the development team into consideration when managing the backlog. They further help to decide whether stories are too big or should be split because of technical dependencies of which the product owner alone cannot be aware.

Similar to the previous issue, a backlog not groomed by collaboration in advance adds unnecessary complexity to the sprint planning event.

6.2.4.5 *Issue: No Testable Acceptance Criteria*

The last issue belonging to the category of bad product backlog management could be treated as a sub-aspect of insufficient story preparation. However, it is treated as a separate problem because the interviews revealed that even when a story seems to be well prepared and thus defines a list of acceptance criteria to be fulfilled after its implementation, product owners regularly fail to specify acceptance criteria in a manner that they are indeed *testable*.

This aspect also becomes apparent when looking at recent survey data, revealing that only one-third of agile projects use test-driven development approaches, whereas even less, namely less than one-fifth, incorporate a behavior-driven approach [271].

6.2.4.6 *Issue: Subsequent Modifications of the Sprint Backlog*

Although explicitly mentioned in the Scrum Guide as something to avoid, case studies revealed that some development teams do not comply with the feature specifications to whom they mutually agreed in the sprint planning meeting. Instead, they modify the set of work on their own as the sprint progresses.

According to Krasteva and Ilieva, this behavior is an indicator of a much bigger problem, which is about questioning the ownership of the product and, hence, the sole responsibility of the product owner [145]. By investigating different teams, they discovered that the reasons for this issue may vary widely. For instance, in their case study, the developers of one team were determined to build a great product, so they changed the stories of the product owner for "better ones," thus undermining his authority by their supreme motivation. On the other hand, another team modified the specifications of the sprint work because of a product owner who had an insufficient technological foundation to clarify the requirements. In both cases, an open and honest process inspection during the retrospective should have led to insights about closer work between the product owner and the development team to guarantee that the whole team understands and commits to what has been specified in the sprint planning meeting.

Experts agree that sticking to the determined feature specification is crucial for proper Scrum implementation because this is what builds trust between the roles. Developers can be sure that their workload is limited and will be approved once it fulfills the acceptance criteria and definition of done, whereas the product owner can anticipate the outcome of the sprint and head for planning the next iteration before the actual one has even ended. Together with proper inspection and adaption of the process, this keeps the overall project flow going.

6.2.5 *Challenge: Sprint Planning*

The overall purpose of the sprint planning event is to *constantly* align the product owner and the development team on what to implement during the next sprint, delivering the highest possible value to the customer.

The emphasis is here on constant alignment because software development is characterized by unpredictable contingencies and the resulting likelihood of rapid requirement changes, for which reason Scrum uses individually planned sprints. In this respect, the particular challenge of sprint planning is to remain focussed on the most value-giving features, laying the foundations for the later sprint work and thus optimizing the whole development flow with regard to these uncertainties.

There are two crucial parts to this workflow optimization. The first one is keeping the team at a maximum yet constant pace, which means aiming for an average velocity (i.e., an equal amount of story points) in each sprint because this allows some forecasting in terms of cycles that are bigger than one sprint only, like planning the release dates of higher-level features that will be implemented over multiple sprints. The second one is balancing this pace against room for process improvement so that the team has a chance to constantly inspect and adapt their way of working.

As will be seen throughout the following sections, the issues presented seem to be caused by this conflict between planning for the increment — delivering as many value-giving features as possible — and planning for team improvement.

6.2.5.1 *Issue: Overestimating the Sprint Scope*

As the interviews and the personal experiences as a product owner revealed, overestimating the development team's capacity for what to achieve during the sprint is not the exception but rather the norm of sprint planning. Of course, this leads inevitably to the situation that a sprint backlog is almost never completely processed at the end of the sprint. Why is this, and what problems might arise?

From the product owner's point of view, the scope of a sprint is deliberately designed to exceed the team's usual velocity because he or she wants to guarantee that there is still enough work left in case the implementation of all other features goes surprisingly well. On its own, this behavior is not problematic, but it certainly gets when the development team is simultaneously *expected* to implement all features of the sprint backlog. This expectation was reported to be often the case and is either caused by an imbalance between the product owner

and development team regarding decision-power (see Section 6.2.2.1 and 6.2.4.2) or a misunderstanding of the sprint goal (see Section 6.2.5.6).

However, it is not only the product owner who contributes to an overestimation of the team's capacity but also the developers themselves, who forget to take everything into account, which simply requires time and, therefore, might affect their ability to deliver. Examples include public holidays during the sprint, team members on sick leave or vacation, time for other Scrum events like backlog grooming, training new team members, and many more.

In any case, there are strong indications that constantly overestimating capacity harms the development team's overall motivation, especially when teams associate unfinished sprint backlog items with failure.

6.2.5.2 *Issue: Dominant Product Owners*

While Eloranta et al. identified "product owners without authority" as a typical anti-pattern of Scrum [269], the conducted interviews with Scrum coaches as part of this thesis also revealed *dominant product owners* as a severe issue.

The problem stems from a false assumption that the development team is subordinate to the product owner's decisions, which is often the case when former management staff become product owners (see Section 6.2.2.1). Typical negative effects of this issue are exerting pressure and pushing the development team to take more tasks than it could realistically handle, as well as last-minute changes that are squeezed into the sprint backlog and which possibly violate the definition of ready (see Section 6.2.5.7) or lead to less slack time of the team (see Section 6.2.5.4).

As already stated in Section 6.2.4, the role of the product owner is critical in arising problems when the corresponding person cannot embrace the core of Scrum, especially the underlying values (see Section 4.4.1). Concerning sprint planning, dominant behavior undermines the development team's authority about the actual sprint work and the members' professional expertise and prerogative to *pick* product backlog items rather than being told which ones to implement.

However, it should be noted that "dominance" does not only relate to product owners with a leadership personality. In the interviews, it was reported that undermining the development team's authority can also happen unconsciously, for instance, when product owners are not able to say "no" to stakeholders and thus make unrealistic promises, which causes more stress for the development team.

6.2.5.3 *Issue: Ignoring Technical Debt*

According to experts, about 15% of resources during a sprint should be spent on tackling technical debt, e.g., fixing bugs or refactoring the codebase [104]. In the Scrum framework, a strategy to reduce technical debt is given by the concept of a definition of done (see Section 4.3), which is a checklist of quality criteria that must be fulfilled for considering an implemented feature as "done" and ready for delivering it as part of the product increment.

While a well-specified definition of done is known to improve code quality [71], product owners are likely to ignore the need for handling technical debt due to their focus on delivering features. And so do they ignore that problems arising from postponing code quality to later points in time will catch up with the team soon. Its future product delivery capability will decrease substantially because of increasing complexity and unclear code dependencies.

This problem can be seen when developers are urged to execute "refactoring sprints." These do not include any features but solely concentrate on tackling the technical debt of the codebase, which has no value to the stakeholders. To avoid turning into an output-focused feature factory, it is, therefore, the responsibility of the development team to demand adequate capacity for tackling aspects of code quality during the sprint.

6.2.5.4 *Issue: No Slack Time*

Similar to not demanding time for assuring code quality by reducing technical debt, the problem of decreasing development performance over time is also affected by over-utilizing the team's capacity during the sprint planning and forgetting to include a *slack time buffer*.

While the Scrum Guide speaks of reserving 10% of the sprint time for mutual backlog grooming, experts agree that there should be at least another 10% of slack time for other collaborative activities, like, for instance, supporting other members of the development team or, doing pair programming. The idea behind this is to ensure that the team can work at a sustainable pace and maintain a healthy environment [226, p. 208] on the one hand, but also to leave room for learning new skills and sharpen cross-functionality of the team.

As Rubin argues, planning the scope of a sprint without considering slack time leads to a situation where everyone solely focuses on his own tasks. This, in turn, enables individual members to become bottlenecks and impede the whole implementation flow. Overall, over-utilization would push developers into a less collaborative mindset, impeding Scrum's core aspect of a self-organizing development team.

6.2.5.5 *Issue: Too Detailed Planning*

From the experiences of the project groups, it became apparent that Scrum novices especially tend to conduct sprint planning sessions with too much detail and upfront specifications, which experienced professionals later confirmed during the interviews.

This is because of a misunderstanding of the "how" part of the sprint planning event (see Section 4.7.1), where developers break the selected user stories further down into development tasks. However, planning every single task of the upcoming sprint in advance is a waste of time. Instead, there should be just enough tasks for developers to start working. Moreover, they can also start *learning* because the sprint backlog is meant to be emergent (on the level of tasks) - just like the product backlog is on the level of stories (see Section 4.5) - to be prepared for technical changes during the sprint that are yet unknown.

Another issue regarding too-detailed planning and time-wasting is when estimation is brought down to the level of tasks. This is of no value to any person directly involved with the development of the product and is usually only done for reasons of accounting for hours. Therefore, it indicates that the team might be embedded in a waterfall-ish environment with the resulting problems that have been explained in Section 6.2.2.

6.2.5.6 *Issue: Not Using a Sprint Goal*

Using no goal for the sprint is a common anti-pattern of the sprint planning event [208]. The problem with this is that the selection of product backlog items degrades into a random assortment of features, providing no cohesion regarding a clear business objective.

According to Pichler, it is crucial for everyone to understand *why* the sprint is carried out [209]. Not only does a sprint goal align the product owner's business objective with the overall product vision, but it also acts as a kind of negotiation between the product owner and the development team regarding what must be achieved at the end of the sprint. In this respect, the sprint goal becomes the mechanism to deal with the problems of decreasing motivation and team commitment due to constant overestimation (see Section 6.2.5.1). It shifts the expectations about a successful sprint away from "implement all items of the sprint backlog" to the fulfillment of the business objective. This business objective should be deliverable even when the development team has not been able to implement the complete feature set of the sprint backlog.

6.2.5.7 *Issue: Ignoring the Definition of Ready*

Adding items to the sprint backlog that do not meet the definition of ready is a popular anti-pattern of Scrum in terms of the sprint planning event and was mentioned by all of the persons interviewed. The development team's rationale behind not rejecting items is the false assumption of an already existing mutual understanding of the story. The missing parts are treated as being obvious and known to everybody so that, for instance, specifying acceptance criteria appears to be redundant.

However, the reality is that unready items will often cause disruptions during the sprint precisely because of arising questions that must be answered by the product owner anyhow. This disturbs the overall development flow and unnecessarily endangers achieving the sprint goal.

6.2.5.8 *Issue: Unfinished Stories Spill over to the New Sprint*

Transferring unfinished stories from the last sprint to the new one is an obvious and often reasonable decision since these items have already been considered to be of the highest value to the customer, which is why they were included in the previous sprint.

The problem, however, is when items "spill over without any discussion," as stated by the professional Scrum trainer Stefan Wolpers [293], because this automatism is possibly a result of what is known as "sunk cost fallacy" [9]. This term relates to a cognitive bias in strategic decision-making, which is known to affect project decisions in software development [64] and describes that humans tend to continue a behavior or endeavor as a result of previously invested "costs," like the donated time, the money spent or the already invested effort.

Therefore, the product owner and the development team must be conscious when transferring items from the last sprint to the new sprint and constantly question whether interim changes to the product backlog - new or modified items - could be more valuable to deliver.

6.2.5.9 *Issue: Not Considering Improvements*

The issue of not including at least one insight as a result of the previous sprint retrospective is, first of all, the responsibility of the Scrum master as the accountable person for aspects of process improvements.

It is particularly mentioned here because it is during the sprint planning when previously identified insights are added to the sprint backlog, making this event the starting point of mutual decision-

making regarding product *and* process-related developments during the sprint.

As an overall impression from the interviews, and as will be explained in Section 6.2.8, it must be said that there is a strong indication of a missing awareness of constant process improvement, which is one of the core aspects of the Scrum framework.

6.2.5.10 *Issue: No Team-Based Decisions*

To live Scrum's values and establish a trustful relationship with highly motivated individuals, it is essential to understand that sprint planning is meant to be a team effort and that everyone's voice must be heard. However, three occasions might impose the sprint forecast as a team-based decision.

The first one is the result of a dominant product owner, as explained in Section 6.2.5.2, who defines the scope of the sprint according to his personal ideas. The second one is when the Scrum team is not free from outside influence, so external forces can impact the planning process. A typical example of this is given by stakeholders (which can be customers or representatives from higher management), who request to take more items after pointing at the team's previous velocity - an attitude that Stefan Wolpers describes as "we need to fill the free capacity" [293]. Finally, there can be "tech leads" of the development team acting as spokesmen and making forecasts on behalf of the other team members. This is especially true for senior developers, who can claim leadership regarding development decisions and might even assign tasks to individual developers.

6.2.6 *Challenge: Daily Scrum*

According to the Scrum Guide, the daily meeting is relatively modest in terms of the underlying rules. The daily basis, preferably at the same time and place, and the limitation to a timebox of 15 minutes are intended to synchronize the efforts of the team members in order to ensure that everybody is aware of how the team as a whole is tracking towards the sprint goal. For this, every developer updates the other team members about what has been achieved since the last daily meeting and if there are any impediments to progress.

Because of this, one would generally not expect many difficulties when holding a daily meeting. However, interestingly, it is this meeting in particular where severe other problems are brought to light, as will be explained within the issues presented in the following.

6.2.6.1 *Issue: No Routine*

This issue describes a scenario in which the daily Scrum meeting does not happen at the same time and place every day or, even worse, is skipped on an occasional basis. As a reason, the interview respondents mentioned the late arrival of one or more participants, no presence because of members working from home, other conflicting obligations of particular persons, and the absence of "leading" persons of the development team or the Scrum master.

Altogether, there is a strong indication that the importance of the daily Scrum is often underestimated, so even relatively small organizational problems can disturb the establishment of a daily routine. On the other hand, the issue also relates to severe misunderstandings of other parts of the Scrum framework, which manifest in not being capable of conducting a fifteen-minute standup meeting every day. This applies, for example, to development teams lacking self-organization or not living the Scrum values (see Chapter 4.4.3), but also to teams with management dependencies (see Section 6.2.2.2) or not enough slack time so that the meeting becomes superfluous since everybody is solely focusing on his or her own concerns anyhow (see Section 6.2.5.4).

6.2.6.2 *Issue: Low Team Spirit*

Since the daily Scrum meeting is meant for the development team to align all members on their way toward achieving the sprint goal, it naturally becomes the place where weaknesses in the overall team spirit immediately reveal themselves in various forms. Some examples that have also been reported by the interview participants are given by Stefan Wolpers in his article about daily Scrum anti-patterns [291].

Disrespect among team members may be revealed when talking with others while someone is trying to share his or her progress with all those present. In addition, it manifests in showing up late or not participating at all, which poses a severe risk to the development team's ability to inspect and adapt the plan toward achieving the sprint goal because of incomplete information. Furthermore, cluelessness, a general lack of interest, and indifference towards an active contribution become apparent when team members are not prepared for the daily Scrum meeting and thus claim not to remember their work status or are unwilling to share their progress. Lastly, spotlight-seekers or persons with a quest for self-glorification may criticize other members, sparking a discussion within the meeting itself (see Section 6.2.6.4) instead of providing helpful assistance once the meeting is over (see Section 6.2.6.5).

6.2.6.3 *Issue: People Don't Listen to Each Other*

A particular issue of the daily meeting mentioned by Moe [181] that also was observed within both ethnographic studies of the project groups was that members of the development team often do not listen to the currently reporting person - even if a strong team spirit is otherwise existent. Unfortunately, there is no clear discernible pattern in the given data. While it seemed at first that this problem depended on individual characters and a person's ability to listen, it later turned out that nearly every group member was affected, sometimes more, sometimes less.

During the project group's process inspection, this issue was analyzed, and people reflected on having lower attention spans when listening about the progress of particular features or elements of the sprint backlog in which they were not immediately involved. As a reason, it was mentioned that people were not able to connect to the work of others when there was not any kind of mental support or visual aid helping to recall what a person talked about in the last daily meeting in order to understand and improve the awareness about his or her actual tasks, how he or she is making progress and how his or her problems might relate to someone's own work.

6.2.6.4 *Issue: Beginning Discussions and Exceeding the Timebox*

Another issue that was observable in both project groups was that members occasionally deviated from answering the three questions of the daily meeting (see Chapter 4.7.2). They reported impediments to their work but immediately began to dive into problem-solving, which was time-consuming because of in-depth discussions. As a result, the participants were often not able to keep the timeframe to the intended fifteen minutes, and other members of the team, not taking part in these conversations, seemed to lose interest and began to talk with each other about topics not related to development [181].

From the data obtained, this problem seems to depend on the participants' overall Scrum experience. While the timebox was exceeded on a regular basis and with stronger deviations from the target at the beginning and when the practical Scrum experience was low, the exceedance rate dropped continuously as each group managed to analyze this issue during their regular process inspections and briefed their Scrum masters to watch for arising discussions carefully. In addition, one group made use of what Scrum experts call a "parking lot" [199] [153], which further improved the daily synchronization of work between members of the development team by simply "parking" topics of arising discussions in a designated section of a whiteboard. That way, the flow of the daily meeting was not affected by conversations

about specific problems that were irrelevant to the rest of the team. At the same time, the Scrum master was able to ensure that interested persons could address parked items after the meeting.

From the answers given in the interviews, this issue also relates to problems described in Section 6.2.4.3 and 6.2.6.2. In case of bad sprint preparation, for example, the development team abuses the daily Scrum as a kind of sprint planning meeting and thus violates the timebox by refining user stories or discussing new requirements. In the event of a low team spirit, the reason might lie in members commenting on every issue or having long monologs.

6.2.6.5 *Issue: Low Willingness to Help*

According to Wolpers, team members experiencing difficulties in accomplishing a task over several consecutive days are a strong indication of deeper problems that manifest during the daily Scrum meeting and stem from the fact that nobody seems capable or willing to offer help [291].

He interprets this issue as either a result of low team commitment, meaning that people may not trust each other or simply do not care for the problems of others, or alternatively as an outcome of bad sprint planning with respect to sufficient slack team (see Section 6.2.5.4).

6.2.6.6 *Issue: The Daily Scrum Downgrades into a Status Report Meeting*

An often misunderstood aspect of the daily Scrum is that it is considered a *status report meeting* from the development team to the Scrum master, product owner, or members of higher management. It goes hand in hand with another false assumption, which is widespread and can even be found in various scientific publications, e.g., by Sharma and Hasteer saying, "The Scrum master also conducts the daily meeting in order to get the status of the project" [251].

However, both are wrong, as explained on the official Scrum website:

"The Scrum Master ensures that the Development Team has the meeting, but the Development Team is responsible for conducting the Daily Scrum. The Scrum Master teaches the Development Team to keep the Daily Scrum within the 15-minute time-box.

The Daily Scrum is an internal meeting for the Development Team. If others are present, the Scrum Master ensures that they do not disrupt the meeting. The Daily Scrum is not a Status Meeting." [247]

There is particular emphasis on this fact because status reports have an obligatory character and usually proceed from lower to higher

management levels within a certain form of hierarchy, hence staying in conflict with the central demand for self-organizing teams in agile work settings. In that sense, treating the daily Scrum as a status report meeting contributes to failing Scrum adoptions [145].

6.2.6.7 *Issue: Command and Control by the Management*

An even stronger derivation from the daily Scrum principle is when the meeting is subconsciously under command and control by the higher management, as explained by Stefan Wolpers [291].

This happens when line managers attend the daily Scrum meeting not just for participation but to gather performance data on individual developers in order to assume control over the team. This behavior relates to the general issue of teams embedded in a waterfall-ish environment and, in this example, defies the very purpose of self-organization. Another example is supervisors waiting until the daily Scrum is over but then reaching out to particular developers to request more specific reporting, which unnecessarily distracts the development team.

6.2.7 *Challenge: Sprint Review*

As written in Chapter 4.7.3, the sprint review is the central learning loop in the Scrum framework for understanding customer needs. It serves as a constant feedback-response cycle for iteratively and incrementally adapting the development of the product and heading it in the most valuable direction.

Based on empiricism, its proper execution is a central component for the correct verification and validation of the sprint result. It thus contributes a lot to the overall quality of the Scrum process.

According to Rubin, the preparation of the sprint review should consist of five actions [226, p. 366]:

1. Determine whom to invite
2. Schedule the activity
3. Confirm that sprint work is done
4. Prepare for demo
5. Determine who does what

At first, the Scrum team should aim to get the right people into the room to obtain the most valuable feedback. A core group is typically invited to every review, but attendees might vary depending on the

implemented features. The team should not forget to focus on specific persons whose feedback is essential for particular aspects of the sprint work and, hence, must be invited, too.

Furthermore, because the sprint review is the only Scrum ceremony that includes participants other than the Scrum team itself, it must be scheduled accordingly. As a result, the determination of the persons who are assumed to provide the most valuable feedback has direct consequences for the scheduling activity because of their higher attendance priority.

The third preparation step is to confirm that what is shown in the review is really done, followed by the fourth and fifth aspects of proper review preparation, namely deciding how the outcome of the sprint will be presented and who will be assigned to which parts of the demonstration.

Altogether, team members must be aware that leaving out one (or more) of these aspects may result in a bad flow during the review session, thus affecting its overall quality and, more importantly, people's perception of the meeting as being something valuable [226, p. 366]. These potential pitfalls will be explained in more detail in the following identified issues.

6.2.7.1 *Issue: No Discussions, No Feedback*

Since the central idea of the sprint review is to obtain feedback and check whether the team is on the right track, possibly the gravest problem is when this feedback is either not received or not incorporated into the next sprint.

One possible reason is that customers are unwilling to participate in the meeting, which has already been discussed in Section 6.2.2.6. However, other issues belonging to the Scrum team itself also contribute to this problem. One example is when the Scrum team has a "we-know-what-to-build" attitude [292]. This may lead to not inviting the stakeholders so that the review is either not conducted at all or not used to discuss the current state of the product with the customers but rather to seek acknowledgment.

As another example, Scrum teams have to deal with participating stakeholders who sometimes may even be passive and unengaged in giving feedback. In this case, passing the reins to the invited persons is necessary to drive the meeting themselves.

6.2.7.2 *Issue: Omitting the Demo for Technical Implementations*

This issue occurs when the development team spends most time of the sprint on architectural work or so-called "glue-code" [226, p. 370] and therefore argues that it is not feasible to demonstrate the results in the review.

At first glance, this seems reasonable, but Rubin clarifies that this assumption is based on an inconsequent implementation of Scrum principles. That is because the product owner must have approved that the team will spend most of the sprint work on technical backlog items during the sprint planning meeting. With this, he must have understood the value of this work and, therefore, should have specified concrete criteria of acceptance, which are essential since it is obligatory that only complete work is presented during the review. In addition, the definition of ready, which every item must pass to be part of the sprint, should contain a check whether the team understands how to demonstrate its implementation. That is why successful architectural work can at least be presented by passing tests, demonstrating its correctness.

6.2.7.3 *Issue: Development Team Is Not Focusing on Relevant Content*

According to personal experience as the product owner of both project groups and from the answers given in the interviews, a common problem of the sprint review is that the development team is over-ambitious in terms of presenting what has been achieved during the sprint and is not focussed on what is relevant for establishing valuable discussions between the stakeholders and the Scrum team.

A typical example is well-prepared slideshow presentations, which fail to take effect because invited stakeholders are not meant to sit and hear but rather to have engaging and interactive discussions with the Scrum team. On the other hand, demoing and talking about every task accomplished is also not the right way. Instead, Wolpers suggests focusing on telling "a compelling story at the beginning of the review to engage the stakeholders" [292], possibly leaving out user stories or tasks that have been necessary but are not in the spotlight of the story being told.

Lastly, many teams seem to violate the concept of "done" and show work that has not been finished during the sprint. Occasionally, there might be a good reason for this, for example, if the implementation cannot proceed further without customer feedback. However, in general, it is not a good idea to include unfinished work because it softens the concept of "done" as a means to ensure code quality and reduce technical debt.

6.2.7.4 *Issue: Treating the Review as Venue of Approval*

First and foremost, the purpose of the review is to inspect the completed work and adapt development based on the feedback obtained. That is why participants in the review not only include the whole Scrum team but, more importantly, invited stakeholders, who can provide valuable comments and inputs to consider for planning the next iteration.

However, some teams misunderstand the review as being the venue for approving or rejecting implemented features, thus degenerating the meeting into a formal sign-off event [226, p. 372] acting as a stage-gate similar to a sequential waterfall process. This is against the definition of Scrum, declaring the review meeting as an informal feedback event, which in turn means that the product owner should approve or reject implemented items of the sprint *before* the actual meeting since he occupies the central role of product leadership and decides whether the implemented features met the definition of done, and as part of that, its specified criteria of acceptance. This process guarantees that only done items are part of the review meeting and prevents discussions about approval by senior-level stakeholders or participants, who otherwise hinder the flow of feedback.

6.2.8 *Challenge: Sprint Retrospective*

In his book "Essential Scrum," Rubin describes the retrospective as "one of the most important [but at the same time] least appreciated practices in the Scrum framework" [226, p. 375]. This assumption is supported by the answers given in the interviews, which revealed that the retrospective is the first of the Scrum events being ignored or left out on purpose. However, underestimating the retrospective's importance for continuous improvement is a serious problem since this puts Scrum's foundation of empirical process control into question and, therefore, endangers the very core of its agile philosophy. In particular, the following issues have been identified.

6.2.8.1 *Issue: No Retrospective or Low Attendance*

According to Rubin [226, p. 392], the issue of neglecting the retrospective stems from organizational problems on the one hand. This happens when people are assigned to multiple teams, so scheduling conflicts could prevent them from attending the meeting. A further example is remote participants who might find attending the meeting inconvenient because of schedule but also due to the fact that

remote participation is generally limited and does not provide the same atmosphere, presence, and feel compared to physical attendance.

On the other hand, there seems to be a lack of awareness or missing value for the retrospective as a tool for enhancing the overall Scrum process and the way people work together as a team. In fact, Rubin identified that people might dissociate from participating by thinking that anything that is not part of their particular working task (like coding or testing) is not worth their time. In his opinion, this disengagement stems from "naiveté regarding Scrum and its focus on continuous improvement" [226, p. 392]. However, he also mentions that an attitude that is quite the opposite could lead to the same problem. This happens when team members believe they have reached the "perfect Scrum implementation," with no room for further improvements and nothing to learn from other teammates or insights about their own work.

6.2.8.2 *Issue: The Retrospective Is Poorly Facilitated by the Scrum Master*

Reoccurring poorly facilitated retrospective meetings have a reinforcing effect on team members with a negative attitude towards collaborative process inspection and give substance to the belief that participating is a waste of time. Rubin gives some examples and mentions "all fluff - no stuff" retrospectives, in which team members are busy and actively discussing but do not come up with concrete insights representing actionable items to consider and work on during the next sprint.

As another example, Rubin has observed that Scrum masters might struggle to take the leadership role in the process. This reveals itself when team members "ignore the elephant in the room," meaning that safety issues could prevent people from bringing up critical problems for discussion, although they obviously have a dramatic effect on the team.

6.2.8.3 *Issue: Depressing and Energy-Draining Retrospectives*

Neglecting the retrospective also stems from repeated negative energy-draining experiences that finally become associated with this meeting. This might be the case when people do not feel comfortable, for instance, when being confronted with results from a bad sprint, which makes them relive the misery [226, p. 393].

Further examples include finger-pointing and mutual recriminations between team members, which is a behavioral pattern that is of utmost importance to be prevented from developing in the first place, or when

retrospectives degrade into therapeutic complaining sessions with a low desire to improve, but to complain about frustrating aspects.

6.2.8.4 *Issue: Incorrect Handling of Identified Insights*

Beforehand, Section 6.2.8.2 mentioned "all fluff, no stuff " retrospectives, in which teams actively discuss but do not come up with concrete things to improve. However, even if insights have been generated and the team also specified actions for implementing these insights, it may fail in two aspects.

At first, Rubin mentions that teams might get overly ambitious and, as a result, set unrealistic goals for improvement. This leads to disappointment once the team realizes that it will not be able to achieve what its members proposed to do.

Second, and according to Rubin, the biggest issue of the retrospective is probably when teams actually generate insights and specify realistic actions for improvement during the next sprint but then fail to implement or work on them. With this, the retrospective itself may be experienced as productive, whereas there is no actual consequence for improvement.

This problem could also be observed in both project groups. While the retrospectives themselves were positive, and the teams were able to generate valuable insights, including concrete actions for the next sprint, the members forgot to actually work on these tasks regularly. During the investigations, it turned out that unconsciously, these tasks have not been considered equal to normal sprint work since implemented features directly impact the sprint outcome and the increment that will be demonstrated in the sprint review. In contrast, work spent on identified process improvements is not as visible and, therefore, felt less valuable.

6.2.9 *Challenge: Understanding Scrum*

For the sake of research clarity, it should be noted that this section about the challenge of understanding Scrum is not based on empirical research but solely on personal experience obtained during the last five years. It therefore does not reflect any scientific results but only a personal opinion.

During market research as part of a European funding project for the software system presented in Chapter 9, there was the chance to speak with more than 50 Scrum masters, all of whom held one or more official Scrum certifications. During these conversations, two things could be noticed over and over again. First, a substantial percentage

of these certified Scrum masters seemed only to have a superficial knowledge of the underlying agile principles on which Scrum is based. Second, the Scrum masters reported that there are sometimes substantial deviations from the ideal Scrum process when theory is put into practice. However, when asked for reasons in more detail, it turned out that the people could not specify them.

The following sections try to give possible answers to both effects.

6.2.9.1 *Issue: Certification as the Single Source of Knowledge*

The institution Scrum.org was founded by Ken Schwaber as an official place for "training, resources, assessments, and certifications to help people and teams solve complex problems" with Scrum [235]. While certification generally is a good concept to assure a certain quality of teaching and likewise to test someone for having obtained profound knowledge, there should be skepticism in terms of the Scrum master certification program.

That is because, from personal experience, there seems to be a misleading trust in the effectiveness of this program, in the sense of a guarantee that a certified person has indeed obtained substantial knowledge. While this is just a personal hypothesis and far from scientifically proven, the following numbers and explanations should at least support this opinion.

The "Professional Scrum Master" certification program (PSM) of Scrum.org is divided into three parts, with independent certifications (PSM I, PSM II, and PSM III) and an increasing degree of difficulty.

All assessments are carried out as online exams. In terms of PSM I and PSM II, the exams consist of multiple choice, multiple answers, and true/false questions, of which 85% must be answered correctly to pass the certification. In the case of PSM I compared to PSM II, significantly more questions (80 vs. 30) must be answered in a shorter time (60 vs. 90 minutes). However, these questions are primarily based on the Scrum Guide, whereas the PSM II questions go deeper and cover two additional focus areas (including the scaling framework Nexus). The most difficult exam, however, is the PSM III certification, which timeframe is 120 minutes and not only requires correctly answering 85% of 30 multiple choice questions but also writing an essay in which a solution to a complex problem from the field must be outlined and solved.

The personal criticism is that even though the PSM I certificate is relatively easy to obtain and is essentially only based on the knowledge of the Scrum Guide, it is nevertheless apparently seen in the industry as a sufficient quality feature for the *professional* implementation of Scrum. This observation is supported by job advertisements for Scrum

masters, which usually only ask for the PSM I qualification, and also by the following certification statistics.

Until the end of 2020, a total of 355,224 PSM program certifications have been successfully completed worldwide. However, when looking at the distributions, extreme differences become apparent, as 345,318 of these certifications fall on PSM I (approx. 97%), 9,032 on PSM II (approx. 2.5%), and only 874 on PSM III (approx. 0.25%) [244].

Accordingly, the "Scrum Master Report" of 2019 showed that 55% of the investigated professional Scrum masters said to have only one certificate. In 53% of all the survey participants, this has been the entry-level PSM I certificate [243].

However, PSM I should be just the starting point for succeeding certifications to guarantee a certain level of quality within the industry and professional Scrum adoption. Maybe for this reason, Scrum.org describes PSM III not only as "highly recognized in the industry" but also as "the only Scrum certification based on testing of knowledge and understanding," which makes it "significantly more valuable than available alternatives" [245].

6.2.9.2 *Issue: Scrum as a Framework Provokes ScrumButs*

The second issue relating to the challenge of understanding Scrum results from the fact that it is considered a *framework*, which opens itself to modifications, as described in Chapter 4.8.

In particular, *framework* means that not everything is prescribed to the smallest detail, so teams still have room for adapting Scrum to their own needs and personal preferences. User stories are an example of that (see Section 4.8.1). Although they are extremely popular, they are not part of Scrum, and teams may decide to use other specification techniques instead.

Besides filling the gaps left out by the framework, there might also be variations to some aspects that actually *are* part of Scrum. For example, it may be better for some teams to neglect estimating items in the backlog. This may seem surprising since estimation is a recommended technique for limiting the risk of being unable to deliver an increment at the end of a sprint, which might happen if items turn out to be more complex than expected. However, estimation alone adds neither customer nor business value to the sprint outcome. Hence, experienced teams may skip estimation completely to further speed up the development process, thus delivering more valuable features within the same amount of time without negative effects.

However, the crux is that leaving room for modifications also carries the risk of making adjustments that are indeed harmful. With the is-

sue of not understanding Scrum properly, chances are high that these teams are just cherry-picking the "simple" elements but neglecting to take full advantage of all the other components, which may need more effort and willingness to implement. As a result, core components are rejected and replaced by workarounds, whereby justification sounds rather like an excuse. This problem is very common [114] and referenced as "ScrumBut" [146] represented by a statement following a particular syntax:

ScrumBut - Reason - Workaround

Scrum.org gives the following examples of ScrumButs: [246]

"We use Scrum, but having a Daily Scrum every day is too much overhead, so we only have one per week.

We use Scrum, but Retrospectives are a waste of time, so we don't do them."

According to the Scrum inventors, a ScrumBut usually results from an arising problem during the adoption of Scrum that seems too hard to fix, so it is retained "while modifying Scrum to make it invisible so that the dysfunction is no longer a thorn in the side of the team" [246]. That way, ScrumButs are known for masking more significant problems whose solution is often inconvenient because of existing work processes.

Therefore, when adapting Scrum, it is crucial to consciously address arising challenges and not eliminate them by ScrumButs as a quick fix. Instead, teams need to assess their agile maturity level since not all modifications to Scrum are generally bad, as described above. In any case, deciding whether a modification is reasonable requires a high degree of self-reflection and honesty about a team's abilities and weaknesses.

6.3 SUMMARY

The following findings can be derived from the results above.

On the one hand, it becomes apparent that the challenges in implementing Scrum are manifold. For example, external factors such as the working environment or the grown company structures can significantly influence the practice of Scrum. This is particularly evident when the supposedly agile process actually corresponds to a waterfall-like procedure, which is reflected, for example, in the fact that plans and documents continue to drive the development process.

However, evolved structures can also be challenging with regard to the Scrum role model. In particular, the role of the product owner

should be mentioned here, whose challenges and problems show strong interactions with other Scrum components. For example, filling this role with previous business analysts or classic project managers can create hierarchies within the Scrum team and give developers less room for their own decisions, e.g., in the form of self-organization of their work. The importance of this role, in particular, should not be underestimated because resulting problems can have far-reaching effects, as illustrated in Figure 6.2 on Page 139. For example, proper ownership of the product is essential to create clear accountability regarding backlog management, counter problems in prioritizing requirements, and establish clear acceptance criteria.

Furthermore, the results clearly show how closely the individual components of Scrum are interlinked because problems in one area can have a direct impact on other parts. For example, insufficient alignment between the product owner and the development team can manifest itself in poor sprint planning, which leads to an overload of the development team during the sprint due to unscheduled slack time. As a consequence, the retrospective can be omitted in order to gain additional time for achieving the sprint goal, which, however, deprives the team of the opportunity for self-reflection in order to better deal with these problems the next time.

In summary, the close interlocking of the Scrum components demands a high degree of caution and personal responsibility from a team so that problems do not build up and thus develop into a negative spiral.

Before an approach for better tool-side support of this interlocking is presented in the further course of this thesis, the following chapter is dedicated to examining the existing Scrum tools to understand if and how current applications address the identified problems.

7

STATUS QUO OF SCRUM TOOL SUPPORT

This chapter addresses the second research question (RQ 2), which asks about the status quo of Scrum tool support. Specifically, and following the results of the previous chapter, the investigations include:

- examining *which tools* are commonly used by Scrum teams,
- how they differ in terms of *offered functionality*,
- and how this functionality *aligns* with the Scrum ruleset and relates to the challenges and issues that have been presented before.

Similar to the previous chapter, details on the respective research methods will be shown first before the results are presented in aggregated form in several individual subchapters representing the different aspects of RQ 2 mentioned above.

7.1 RESEARCH METHOD DETAILS

As shown in Table 7.1, a combination of three research methods was chosen to investigate the different aspects of RQ 2 from various angles.

RESEARCH METHODS	SUBJECT MATTER
Literature Review	Surveys Scientific Papers
Field Studies	PG MAGIC PG MAGICIAN
Usability Analysis	Heuristic Evaluation

Table 7.1: Research methods for investigating the status quo of Scrum tools

First, a *literature review* should identify and gather data about the landscape of tools that Scrum teams use to organize their development work. This is followed by two *field studies* where some of these tools have been used in practice to gather hands-on experience while comparing their functionality against the Scrum ruleset. Lastly, these findings have been combined with results from a *usability analysis*, in which a selection of different tools, particularly designed for manag-

ing agile Scrum projects, have been investigated regarding usability problems concerning the previously identified challenges and issues presented in Chapter 6.

7.1.1 *Literature Review*

Two types of literature have been used to investigate how Scrum teams incorporate different tools to organize their development lifecycle and understand the landscape of tool variety.

Agile surveys

The first type represents *statistical surveys* providing information on a global scale on general aspects of agile adoption or Scrum practices in particular. One survey with particular interest for this thesis is the "Annual State of Agile Survey" conducted by CollabNet VersionOne, which, after its start in 2007, has emerged as the largest, longest-running, and most widely cited agile survey in the world [271]. Every year, the survey investigates both the challenges faced by organizations jumping on the agile bandwagon and the benefits that the agile development process has brought them. It also explores the usage of agile principles and specific tools as well as incorporated techniques for scaling enterprise agility. By gathering data annually from all over the world, the survey aims to "provide insight about the adoption of agile; what new practices are emerging; and how the culture of agile is changing" [270]. Behind this background, it is important to know that the survey is not consistent over the years but shows certain variances in methodology, and hence, comparisons between data sets of different years have limited comparability, which must be taken into consideration while interpreting the data. Another point of criticism is that the reports mainly show results in percent value, so whereas the overall numbers of respondents are transparent to the reader, it remains unclear for individual questions whether respondents may have skipped it. However, for identifying which types of tools are used by agile teams, the reports are well-suited and clearly show unquestionable results.

Scientific papers

The second type of literature represents *scientifically published papers*, which, in contrast to surveys, focus on specific questions about agile tool support. However, it must be said that research in the area of agile tool support is widely unexplored. In particular, it is more focused on providing selection mechanisms of existing tools to help Scrum teams in decision-making, e.g., [261] and [275], instead of analyzing whether the tools existent are suitable and built according to the agile values or the rules of the Scrum framework. In this respect, it was surprising that not a single paper about usability aspects of tools and their consequences for agile adoption could be found, which is why this thesis is apparently the first to do so.

7.1.2 *Field Studies*

As explained in Chapter 6.1.2, the project groups MAGIC and MAGICIAN were used for field studies and to gain hands-on experience about how various tools integrate with the Scrum development process. Each group used the same set of tools, which are as follows.

The first is the most basic analog toolset, *pen and paper*. The product backlog consisted of handwritten index cards representing user stories. The sprint backlog was implemented as a task board, using a whiteboard and sticky notes for the individual tasks, which that way could be easily moved between the columns, while the whiteboard allowed to add handwritten notes for special notations, such as annotating impediments or temporal relations between tasks.

Second, the project management software *JIRA* was chosen for two reasons. First, this tool obtains clear market leadership and is the most widely used and established application for managing agile software development projects [270]. As a second reason, the interviews, as well as personal discussions with Scrum coaches, gave a strong indication that despite the fact of claiming market leadership, Jira is explicitly not recommended by many agile experts but is seen as a "necessary evil" [200] because of missing other digital tool alternatives. Because of this apparent contradiction, Jira was considered interesting to investigate.

Lastly, the tool *ScrumDo* was chosen for comparing Jira's functionality against a new competitor and promising application, which was advertised for including novel features explicitly designed for Scrum.

In the case of PG MAGIC, the students worked on different topics over the course of one year. Hence, tools could be switched in between, thus allowing a natural break and fresh start between the independent sub-projects. As a result, each tool was used on average for four months. This was different in the case of PG MAGICIAN, which had only the single project goal of developing and evaluating a novel Scrum tool by incorporating it into the group's own development process from the very beginning. For that reason, the three tools investigated were used for only one month each during PG MAGICIAN's initial phase of Scrum adoption.

For collecting data, the regular process inspection interval was used, i.e., both project groups decided to include a fixed "tool review" into the sprint retrospective, thus elaborating concerns about the user experience and usability.

7.1.3 *Feature Analysis and Heuristic Evaluation*

The field studies identified limitations that formed the basis for the last component of the investigation of the second research question, which was a profound *feature and usability analysis* of four digital software development management tools, as shown in Table 7.2.

TOOL	WEBSITE
Jira	www.jira.com
ScrumDo	www.scrumdo.com
VersionOne	www.collabnet.com
IceScrum	www.icescrum.com

Table 7.2: Investigated Scrum tools

For the same reasons as during the project groups, *Jira* was selected as being the market leader and *ScrumDo* for being a promising competitor with features that have been particularly designed for Scrum. In addition, *VersionOne* was chosen because it is a widely used agile project management tool and regularly gets the highest recommendation rates in the annual "State of Agile Reports" [271]. Lastly, *IceScrum* was chosen because, on the one hand, it represents a European representative among the otherwise American vendors and, on the other hand, it has positioned itself as a direct competitor to Jira, but with the distinction of being a "true agile project management tool." Therefore, among other possible candidates, it was the most promising for the analysis to derive a status quo of tool support for Scrum.

The analysis used a fictional scenario to simulate a software development project

As preparation for the tool analysis, a *fictional project scenario* was created, simulating the development of a mobile shopping list application based on a simple set of requirements. As shown in Table 7.3 , the core functionality of this mobile application is assumed to only consist of managing various shopping lists, which means that the future user can create/delete lists and add/remove items to/from those lists.

Notably, this scenario shows that the development of even this simple set of features must be *planned* because of temporal relations between different development requirements. For instance, it would make sense to work on *frontend* aspects at first and start to design UI mockups because visual layouts with different navigational flows usually have consequences on the optimal way to store and retrieve data from the database, hence may affect the information architecture. The same holds for the selection of the UI framework because various frameworks work best with certain types of databases for the *backend* of the application. *Testing*, on the other hand, depends on the kind of test. For instance, usability testing regarding the user interface can

ASPECT	REQUIREMENT
Core functionality	Create shopping list Delete shopping list Add item to shopping list Remove item from shopping list
Frontend	UI design mockup and navigation flow Framework selection
Backend	Database setup Information architecture
Testing and publishing	Conducting user tests Publishing the app in stores (Android, iOS)
Visionary features	Location-based notifications about deals Product availability and nearest stores

Table 7.3: Requirements of the simulated mobile shipping list project

be done upfront in any development work using throwaway paper prototypes. In contrast, software testing requires at least parts of the real system, e.g., small individual units (unit testing) up to the whole implemented system (acceptance testing). Lastly, the scenario also shows that, typically, development is never complete. There is always something to improve or *visionary features* to implement in the future. In this example, the mobile shopping list application could be enhanced by providing location-based notifications about special deals at stores near the user's current location. As another example, a shopping list may visualize the availability of products and suggest the nearest store to buy certain items from the list.

Using this scenario as a starting point for the investigations of the four applications, the feature analysis started by simulating actions of a common sprint cycle scenario, including:

1. the necessary steps for *preparing a sprint and managing requirements*, such as creating user stories in the backlog, grouping them, specifying acceptance criteria, attaching additional material (e.g., images) to a story, as well as prioritizing stories according to the Scrum rules concerning proper management of the product backlog,
2. actions of the *sprint planning* event, such as displaying the backlog in a group setting, estimating stories with story points, selecting stories for a sprint, defining a sprint goal, creating tasks and assigning developers,

3. digital support of the *daily meeting*, in which each developer answers the three questions "What have I done yesterday?", "What will I do today?", and "What are my impediments?",
4. possibilities to *groom the backlog*, so that developers and the product owner are supported in refining stories collaboratively,
5. support of the *sprint review* event by presenting what has been achieved in the sprint and collecting new requirements for the next iteration,
6. and features for conducting a *sprint retrospective*, such as collecting data like emotions or personal opinions about certain aspects of the development process, defining insights and process improvements, and considering them for the next sprint during the succeeding sprint planning event.

Heuristic evaluation

While paying attention to comparing the offered functionality to the Scrum issues and challenges presented in Chapter 6, the usability analysis was conducted with *heuristic evaluation* as initially presented by Jakob Nielsen and Rolf Molich [193]. In this "discount" inspection technique, usability experts evaluate user interfaces against heuristic principles, representing "rules of thumb" as a well-established and proven set of guidelines that tend to result in good interface design.

Since Nielsen had shown that the percentage of found usability problems ranges only between 20% and 51% per individual evaluator, the findings from four evaluators[1] have been aggregated. According to Nielsen, this is the number with the optimal ratio of benefits to costs and sufficient to identify the most significant problems because of the logarithmic nature of the proportion of usability problems found by various numbers of evaluators (see Figure 7.1) [188][192].

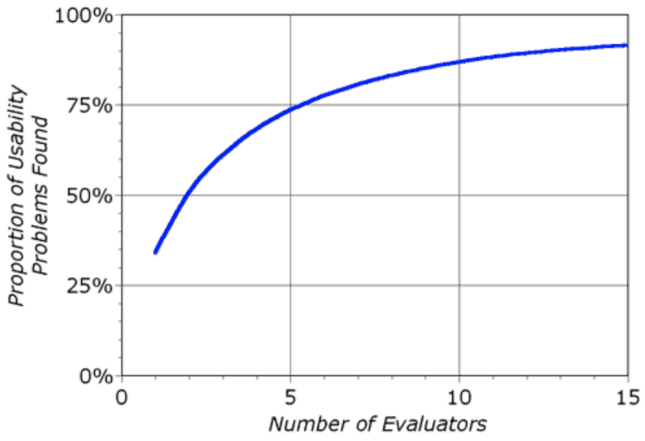


Figure 7.1: Heuristic evaluation: costs vs. benefits [189]

1 Myself and three selected computer science master students with HCI focus area and profound knowledge about usability inspection methods.

The Nielsen heuristics

For the inspection of the four Scrum tools, Nielsen's modified set of 10 heuristics was used [189], which were initially published as follows:

1. *Simple and natural dialogue*: Dialogues should not contain information that is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility. All information should appear in a natural and logical order.
2. *Speak the user's language:* The dialogue should be expressed clearly in words, phrases, and concepts familiar to the user rather than in system-oriented terms.
3. *Minimize the user's memory load:* The user should not have to remember information from one part of the dialogue to another. Instructions for the use of the system should be visible or easily retrievable whenever appropriate.
4. *Consistency:* Users should not have to wonder whether different words, situations, or actions mean the same thing.
5. *Feedback:* The system should always keep users informed about what is going on through appropriate feedback within a reasonable time.
6. *Clearly marked exits:* Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue.
7. *Shortcuts:* Accelerators - unseen by the novice user - may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users.
8. *Good error messages:* They should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
9. *Prevent errors:* Even better than good error messages is a careful design that prevents a problem from occurring in the first place.
10. *Help and documentation:* Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, be focused on the user's task, list concrete steps to be carried out, and not be too large.

Since heuristics are not necessarily rigid and allow a certain degree of interpretation, this evaluation technique is an *informal method* of usability analysis. Therefore, it may deliver different results than an empirical testing of the software with real users. For instance, Law and Hvannberg have compared the effectiveness of a heuristic evaluation

with usability testing and concluded that heuristic evaluation will find between 30% and 50% of problems found in a concurrent usability test [156]. However, while Jeff Sauro has echoed these findings, he has also shown that heuristic evaluation will, on the other hand, uncover the *most common* issues (in his report, he speaks about 100% of the top ten most common problems and 75% of the twenty most common issues), thus making heuristic evaluation a valuable method to deliver quick insights about fundamental usability concerns.

7.2 RESULTS

The following sections present the results obtained from the different studies explained above.

7.2.1 *Tool Types and Usage Trends*

First of all, tool support for agile teams can be divided into two main categories, which are *analog* and *digital* tools.

Analog tools

Analog tools utilize physical objects, commonly index cards with handwritten notations, for specifying product requirements, e.g., in the form of user stories. As a result, the product backlog is represented by a prioritized stack of index cards, whereas the sprint backlog is usually visualized as a task board in which tasks are represented by sticky notes that are associated with particular stories that previously have been selected for the sprint (see Figure 7.2).

Figure 7.2: Analog task board[2]

2 Source: https://medium.com/@sashabondareva/scrum-task-board-offline-or-online-b341719fa472

This choice is because user stories contain more information, like a title describing what the story is about, a short description in the user story format, acceptance criteria, or notes added to the story to clarify details. In contrast, a task usually consists of a single sentence description plus an assignment to one or more team members. In addition, sticky notes are particularly suitable because tasks must be moved between different columns of the task board, depending on the actual state. Therefore, attaching and re-attaching is well supported because of the stickyness of the note.

Due to their flexibility and since analog tools are ubiquitous, they are still the first choice of many Scrum experts. However, there is a clear downward trend. While 44% of agile projects used analog tools in 2013 [270], this number continuously decreased to 28% in 2018 [271]. Given these numbers, it is important to know that analog tools are seldom used alone but rather in combination with digital tools, which will be explained in the following.

Digital tools

In contrast to analog tools, digital tools of agile Scrum teams can be further divided and broadly fall into the following categories:

- general-purpose office tools (spreadsheets),
- application lifecycle management,
- agile application management,
- and lightweight web-based project management tools.

Spreadsheet applications

Spreadsheet applications run locally, like *Microsoft Excel*, or within the cloud as web applications, such as *Google Docs*, whereby the cloud versions allow a simplified collaboration because of shared documents between different users. Since spreadsheets are general-purpose applications initially designed for different calculation purposes in the context of office work, teams usually use templates to incorporate the necessary features for managing the Scrum process. These range from being very simple, like the one presented by Mike Cohn [58] (see Figure 7.3), to sophisticated commercially available documents supporting complex reports and process analysis features.

From a user experience point of view, spreadsheets are characterized by low interactivity. While documents can be customized to a great extent (especially by using macros), everything has to stay within the range of the spreadsheet metaphor, consisting of lists and tables. As a result, the application logic of the process template is limited to the logic of the spreadsheet application itself, so user inputs are not provided by interactions like button clicks or drag-drop functionality but primarily via textual data operations.

ID	Theme	As a/an	I want to ...	so that ...	Notes	Priority	Status
2	Game	moderator	Create a new game by entering a name and an optional description	I can start inviting estimators	If games cannot be saved and returnd to, the description is unneccessary	Required	done
3	Game	moderator	Invite estimators by giving them a url where they can access the game	We can start the game	The url should be formatted so that it's easy to give it by phone		done
5	Game	estimator	Join a game by entering my name on the page I received the url for	I can participate			done
6	Game	moderator	Start a round by entering an item in a single multi-line text field	We can estimate it			done
8	Game	estimator	See the item we're estimating	I know what I'm giving an estimate for			done
~~40~~	~~Game~~	~~participant~~	~~Always have the cards in the same order across multiple rows~~			Replaced with A08 because I didn't want the story to talk about „the same order" as that might be a UI implementation detail	~~Todo~~
35	non-functional	user	Have the application respond quickly to my actions	I don't get bored			done
36	non-functional	user	Have nice error pages when something goes wrong	I can trust the system and its developers			done

Figure 7.3: Spreadsheet product backlog [58]

Despite these limitations, agile teams widely use spreadsheet applications, with a slight downward trend over the last five years. With a three-quarter share in 2014 and 2015, the value decreased to a share of approximately two-thirds in 2018 [271] and one-third in 2022 [300]. However, the annual state of agile surveys did not investigate the actual use case of spreadsheet application within agile teams, which means it remains unclear whether the investigated teams use spreadsheets as the central component for managing the whole development process or just for particular aspects, like for instance calculating project budgets. Nevertheless, what can be shown is that each year, spreadsheet applications out of all tools mentioned by agile teams have by far the lowest rate of future usage plans and belong to the tools with the lowest satisfaction or recommendation rate [271].

ALM applications

In contrast to spreadsheets, *application lifecycle management* (ALM) applications are much more specific in terms of providing extensive features for tracking and organizing the project management process, such as planning resources and scheduling of tasks, monitoring and visualizing progress, e.g., through Gantt charts, as well as rich functionality for status reporting or data analysis.

As identified by agile surveys, *Microsoft Project* and *Microsoft TFS*[3] (see Figure 7.4) are the most commonly used applications of this category and are used by approximately one-quarter of agile teams [271].

MS Project is a standalone application that strongly resembles *MS Excel*, not only in terms of the graphical user interface design but also in terms of the user experience and spreadsheet-like interaction.

3 Nowadays known as "Azure DevOps Server"

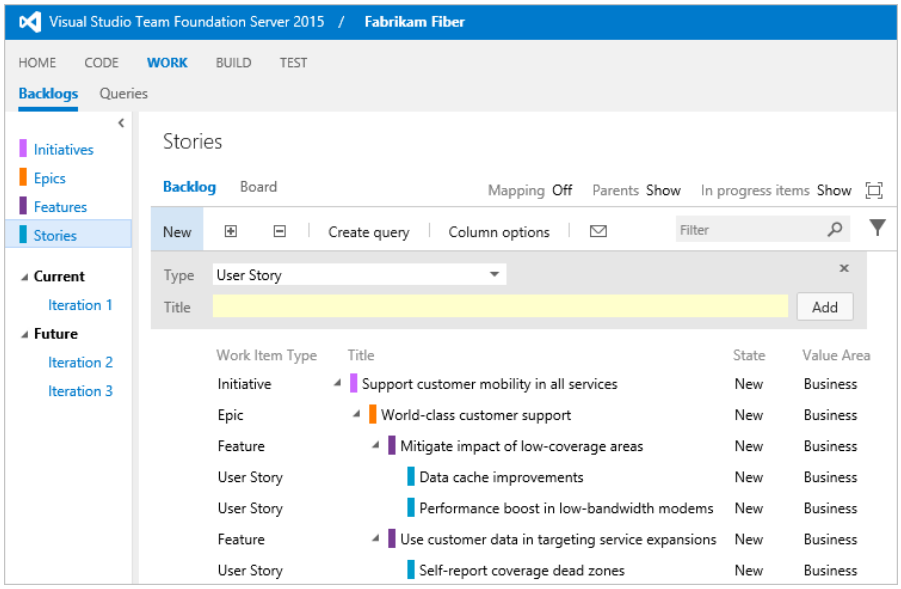


Figure 7.4: Microsoft TFS [178]

In contrast, Microsoft TFS is not a standalone application but a software platform that provides different services and connects, for instance, MS Project to other applications of the Microsoft software development environment. As an example, this integration allows to associate programming code written in the IDE *Visual Studio* with tasks that are managed by MS Project and also to include these references as status reports in source code management modules of TFS to establish a link between data of the project management tool and the version control system.

From the user experience perspective, ALM applications such as Microsoft TFS are characterized by complex functionality and a rich but cluttered feature set often provided through connected other services. Because of this, these tools require more significant configuration effort and have a steep learning curve. Hence, users need a certain amount of training before using them.

Agile ALM applications

The third category of digital tools is *agile ALM applications*, which - as the name suggests - differentiate from traditional project management applications in terms of claiming to offer an agile-specific set of features, which is, for instance, characterized by functionality for enhanced team collaboration and self-organized management of tasks. When looking at these applications, it is interesting to see that some of them are promoted as "general agile" tools (e.g., Jira, VersionOne, CA Agile Central), whereas some others address a particular methodology (e.g., GitScrum, ScrumDo).

Of all agile project management tools, *Jira* (see Figure 7.5) is by far the most frequently used application for managing the development process and is utilized by 65% of all agile teams [271].

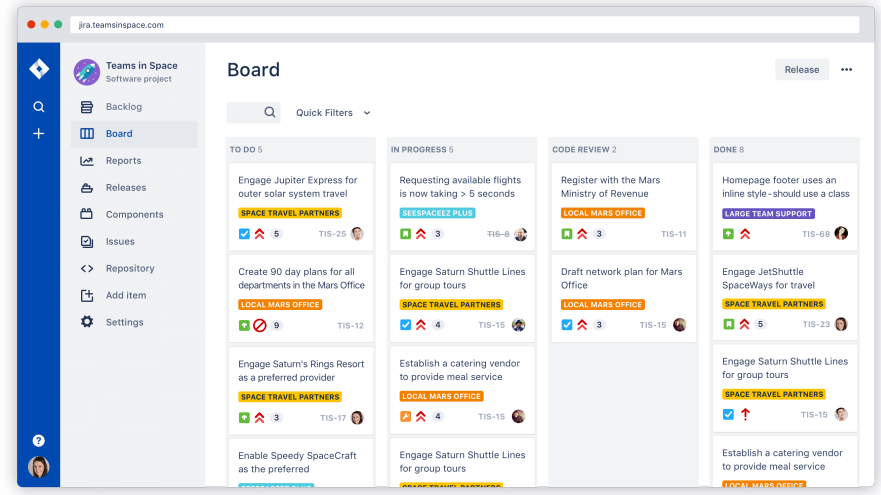


Figure 7.5: Jira[4]

Moreover, the annual State of Agile survey identified Jira and VersionOne as the tools with the highest recommendation rate. Therefore, because these tools have been explicitly designed for supporting *agile* processes, this category has been the focus of research concerning the status quo of Scrum tool support for this thesis.

Lightweight project management apps

Lastly, another category of tools emerged over the last years and can be referenced as *lightweight project management web apps*. These applications are characterized by a simpler set of features and, thus, do not provide a solution for all but only for particular aspects of project management with aesthetically pleasing and modern user interfaces.

Typical representatives of this category are applications like *Trello* (see Figure 7.6), *Asana*, and *Monday*, which are all concerned with managing user tasks.

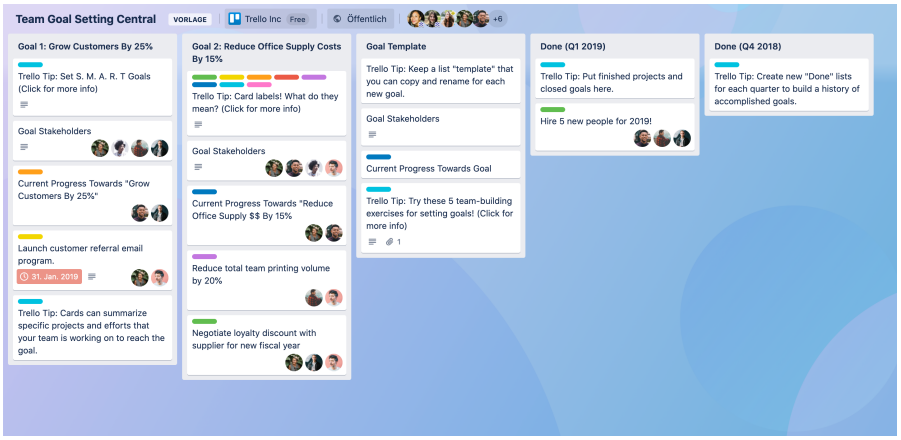


Figure 7.6: Trello[5]

4 Source: https://www.atlassian.com/de/software/dev-tools
5 Source: https://trello.com

In this connection, the objective of Trello is to use *boards* consisting of freely definable columns representing statuses and containing cards that may represent any arbitrary type of item. With this flexible concept, the application applies the concept of "cards in columns" to a wide range of scenarios. For these, it also provides many templates, e.g., for tracking interview studies, managing publishing processes, conducting heuristic evaluations, and even wedding planning. Among many others, there are also templates for agile contexts, such as sprint backlogs visualized as task boards.

In contrast to Trello, Asana and Monday (see Figure 7.7) do not make use of cards that are moved between different columns but instead focus on highly customizable *lists* and *tables*, which also may be used for a wide range of application scenarios.

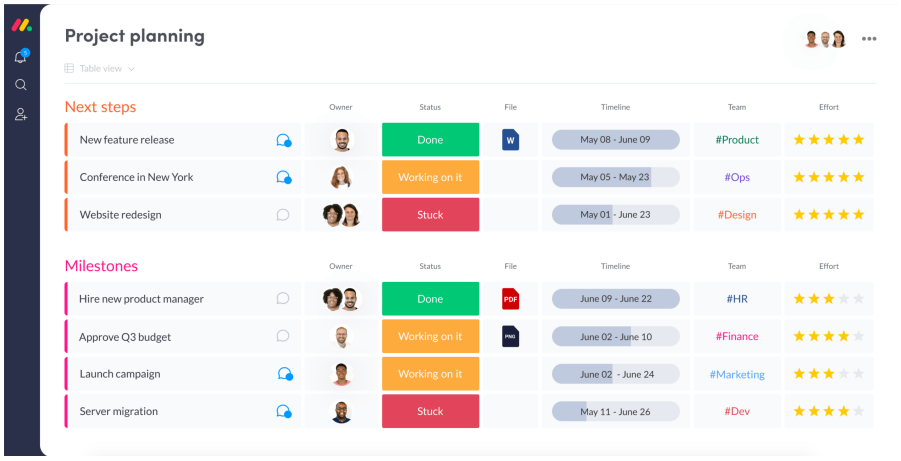


Figure 7.7: Monday[6]

Regarding suitability for specific tasks, these kinds of applications are a two-edged sword. On the one hand, they provide great flexibility due to their customization options. On the other hand, there are clear limitations because of the general-purpose design, which is nonetheless restrictive in terms of the underlying architecture and design metaphor. For this reason, these applications may not have been identified by surveys and reports as commonly used by agile teams. Nevertheless, they are mentioned here because, in the case of the conducted interviews as part of this thesis, many of the respondents referred to actually using these types of applications for specific tasks, especially in situations when the main process management application is either not supporting a particular use case at all or provides an unsatisfactory user experience so that teams want to gain benefit from the applications' simplicity.

6 Source: https://monday.com

7.2.2 *Limitations of Today's Agile ALM Tools*

What features do agile ALM tools offer? Which parts of the Scrum framework are covered? More importantly, are there any aspects that are missing?

These questions have been the rationale behind the feature analysis and heuristic evaluation of four agile ALM tools (see Section 7.1.3), from which two have also been tested in practical application during field studies in the project groups.

Based on these findings, this section will present usability concerns and fundamental limitations regarding the given features and how they contradict the given Scrum rules and agile philosophy. By drawing connections to the previously identified challenges and issues of Scrum, a status quo of Scrum tool support will be derived and presented at the end of this chapter.

7.2.2.1 *Terminology: The Importance to Speak the User's Language*

Violations against the Nielsen heuristic "Speak the user's language" [193] have mainly been found in Jira, VersionOne, and ScrumDo. In this context, it is essential to know that the gravity of inappropriate language depends on the actual consequences for the user. For instance, it may result in relatively small usability problems affecting how the user can solve a particular task, but moreover, it may also give a false picture, which is misleading him or her on a more substantial level of understanding.

For example, ScrumDo does not use the term "backlog item" but instead speaks of "cards" as items in the backlog. This is considered a minor problem because of how a backlog is managed without digital tool support, which is by using physical cards, as explained in Section 7.2.1. Thus, although "card" is not Scrum terminology, it is well-known and actually may help former analog Scrum team members orientate themselves. However, the more important point is that the term "card" is descriptive and has no evaluative character.

In contrast, Jira describes all items of the backlog as "issues," which is a term with a negative connotation[7]. In this case, the violation of the heuristic can be considered a severe problem because treating backlog items as something inherently problematic conveys the wrong message and is against the core agile value to actually *welcome* changes of product requirements.

These examples should show that not speaking the user's language may lead to minor, but also to major usability problems [187] and

7 The term "issue" stems from Jiras background as a bug tracker application.

therefore should be avoided carefully. Although wrong terminology may have no impact on persons having considerable Scrum experience, it may nevertheless unnecessarily affect novices on a subconscious level when trying to learn the agile philosophy. However, it should be noted that this assumption is based on personal experience and not on scientific evaluation. However, it stems from witnessing hundreds of oral student exams, where the results clearly showed that students using correct terminology also better understood the underlying relationships and rules of the subject matter. In contrast, students who had not internalized specific terms often could not explain facts and important coherences. Hence, the same could be true for learning Scrum properly. Therefore, using the correct Scrum terminology is considered important.

7.2.2.2 *Complexity and Consequences for Agility*

Three of the four agile ALM tools showed massive complexity during the evaluations, which led to long familiarization periods. This applies to Jira, VersionOne, and ScrumDo. Behind this background, many violations against the heuristic "Simple and natural dialogue" were found (more than 20 per application). However, since describing each violation would not be helpful for this thesis, the following examples shall be enough to illustrate the resulting problems.

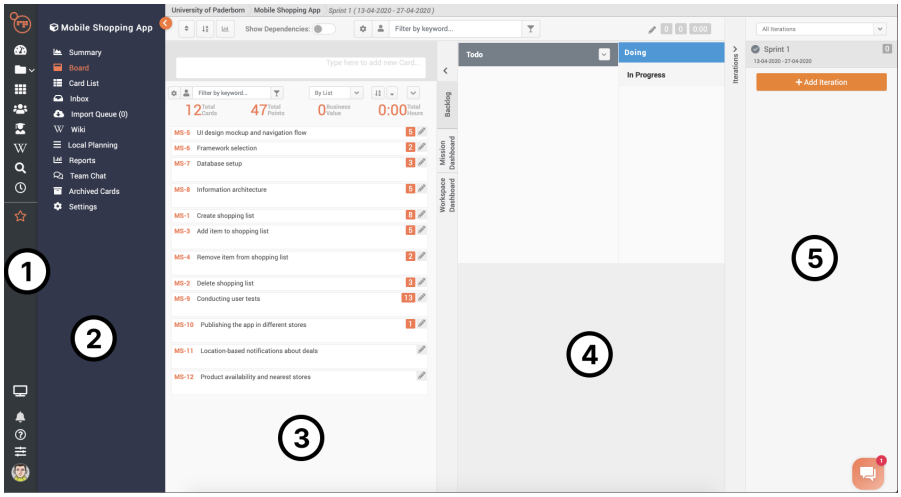


Figure 7.8: ScrumDo

To begin with, Figure 7.8 shows the main view of the ScrumDo application. Even on the very top level, the user interface shows significant problems regarding a very nested layout and overall structure. From left to right, it begins with a small global menu bar (1), a wider project-related menu bar (2), a collapsable area offering functionality for the backlog and two types of dashboards (3), the main content area (4) and a collapsable area showing a list of iterations (5).

Two main aspects prevent this structure from being a simple and natural dialog. The first relates to *clustering the navigation space* into comprehensible semantical chunks or coherent information groups. For instance, the small menu bar on the left (1) mainly contains global functionality, such as a search function, a feature for managing uploaded files, and a global wiki. However, this schema is not consistent because it also contains a "story mapping" feature (the third from the top), which is a technique invented by Jeff Patton [205] as an alternative to a product backlog, better suited to provide a "big picture" of a project by illustrating dependencies and connections between user stories [206]. The critical point to recognize here is that story mapping is not on a global level but related to a specific project and, hence, should be placed in the project menu bar (2). Along the same line of reasoning, "Backlog" should also be part of the project menu (2) instead of extracting it into a separate area (3), which is especially hard to read because of its vertical menu.

Second, the main user interface is also problematic because of underlying and incomprehensible *interrelations*, which also relate to problems of the "Consistency" heuristic. For instance, clicking on "Summary" in the project menu (2) makes the "Board" menu entry disappear, which can only be brought back by clicking on an item of the iteration list (5). Since list (5) could have collapsed to the right edge of the screen, something trivial, such as accessing the board, may become a very challenging task.

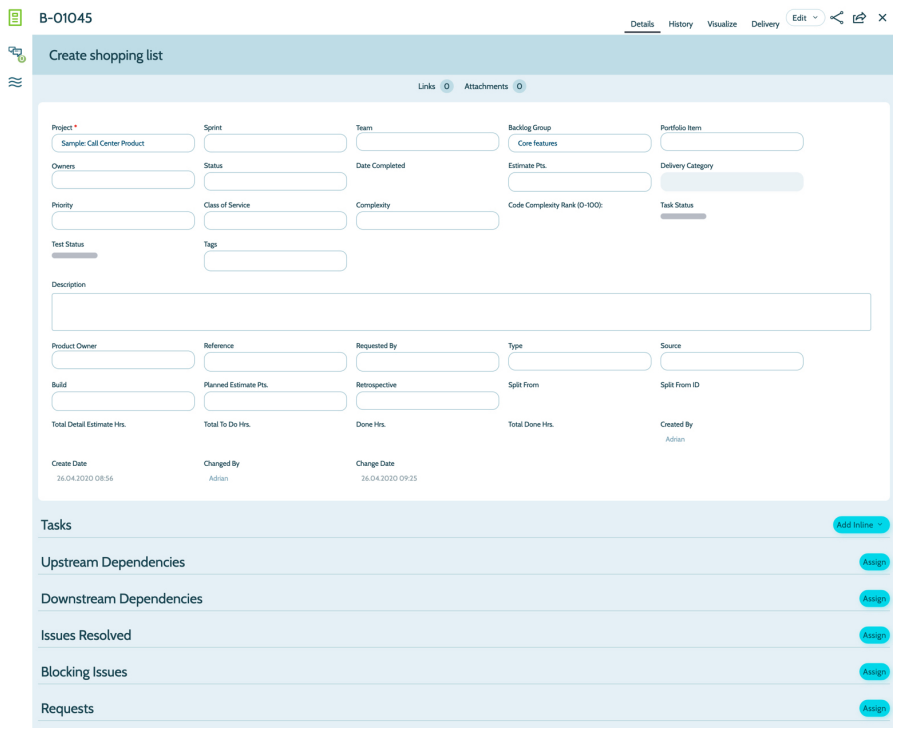


Figure 7.9: VersionOne

As a second example, Figure 7.9 shows the user interface of VersionOne, which opens after the user clicks on a backlog item to see its details. In this context, it is crucial to understand that looking at an item's details is an essential feature of any agile planning application because these details are continuously needed and accessed during all sprint activities.

The top half of the image shows the "essential" details of an item. At least, that is what a user would expect since the white area is visible at first sight, whereas other details are clustered below that area into individual subgroups, all of which are initially collapsed and must be opened before accessing the given information. However, apart from the fact that very relevant information is initially hidden (such as the tasks), even by just considering the top half, this dialog is far from being simple or natural for several reasons.

At first, the user is likely to be overburdened by the sheer amount of input fields, which count up to 24 plus one hidden field (because the title is editable by clicking it, which reveals another input field). On closer inspection, it turned out that many of these fields are unnecessary, either because of over-specification, which is something to avoid in agile planning, or because of redundancy. For example, there is no need to specify three values for estimating an item's complexity, like in this case by using "Planned Estimate Pts.," "Complexity," and "Code Complexity." Instead, it is intended for agile planning to use only *one* single and abstract value of estimation [56].

Second, there is hardly any relationship between coherent information. For instance, the three input fields dealing with the item's complexity do not appear next to each other. Likewise, there are several input fields for specifying some sort of ownership or personnel assignments such as "Team," "Owners," "Product Owner," and "Requested By," none of which seem to be related to any of the others because of missing coherent groups.

In the context of grouping, the cognitive overload of the user is also caused by violations against the *law of proximity* (see Figure 7.10). This law is part of the "Gestalt principles" [138] and describes how the human eye perceives relations between elements that are visually close to each other, whereas item sets that are separated from each other, e.g., by whitespace, are perceived as unrelated.

As a result, the human eye identifies two groups of items in Figure 7.10, one at the left and one at the right, whereas in the case of the dialog in VersionOne, everything is near each other so that groups cannot be identified by visual perception.

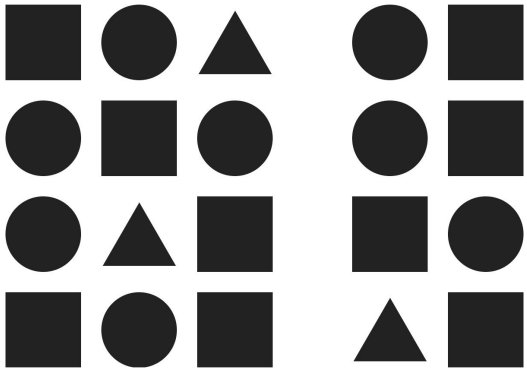


Figure 7.10: Law of proximity[8]

Furthermore, the user interface of VersionOne (see Figure 7.9 on Page 186) shows many other usability problems, such as disregarding a logical order. For example, it is incomprehensible why the description of the item is not directly placed after the title but instead right in the middle of the overall flood of other information. Moreover, the heuristic "Speak the user's language" is violated by, among others, prominently showing the internal ID of the backlog item at the very top, which has no meaning for the actual user.

These examples illustrate that many of the identified usability problems arise from the sheer complexity of the features provided. More features generate larger dependencies, which usually manifest in cluttered user interfaces. For instance, Jira provides a tremendous amount of customization options. However, the user interface must be flexible enough to handle a huge set of various things to display.

Regarding the feature sets provided by Jira, ScrumDo, and VersionOne, large parts of the functionality must be questioned as to whether they are helpful in agile development contexts. Features like time tracking of tasks, sophisticated hierarchies of requirement specifications, detailed pre-ahead planning, complex progress measurements, and control mechanisms all stem from classical project management and the era of sequential development models. Agility, however, is characterized by getting rid of anything unimportant to deliver what momentarily has the most value for the customer. As a result, the fundamental shift from sequential to agile development models should also be reflected in a feature set that breaks the chains from traditional approaches and can support the essentials of Scrum or agile values in general.

8 Source: https://www.chrbutler.com/gestalt-principles-of-design-proximity

7.2.2.3 *Backlog Management*

In Scrum, everything revolves around the sprint as the central element of the iterative development process, at the end of which an increment of the product is potentially deliverable. Among a team of skilled and professional developers, it is first of all indispensable that the backlog is thoroughly prepared by the product owner to ensure that everybody will continuously work on the most valuable items.

Surprisingly, the investigations revealed that well-established applications lack basic but essential aspects for preparing a sprint and managing requirements. The following will illustrate this limitation using the market leader Jira as an example.

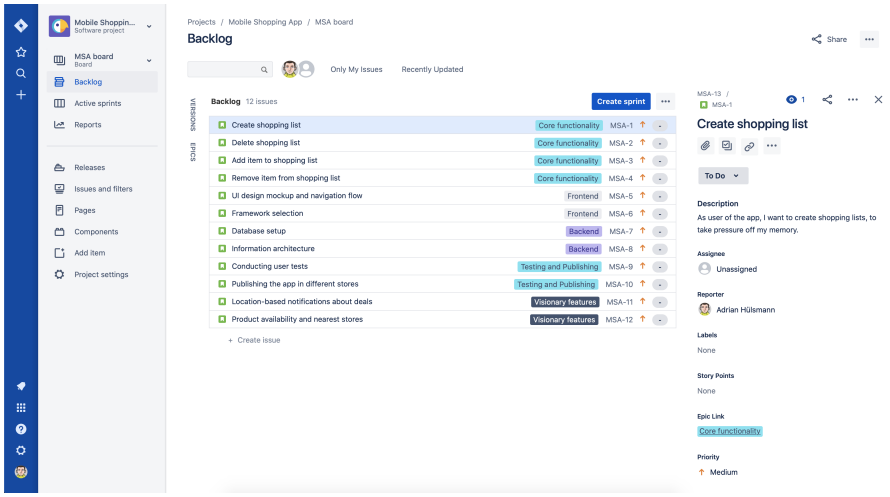


Figure 7.11: Jira backlog

Figure 7.11 shows Jira after the requirements of the fictitious project scenario (see Table 7.3) have been specified. The backlog is shown in the middle, whereas details of a selected item are displayed on the right side, which is shown enlarged in Figure 7.12 on the next page.

As can be seen, Jira does not support the specification of acceptance criteria, nor does the application provide the possibility to check the specification of an item against a definition of ready. This circumstance has severe consequences because acceptance criteria are the only concept for verifying the sprint outcome. Equally, the definition of ready (see Chapter 4.5) serves as a control mechanism for the sprint planning event to ensure that selected items fulfill a certain specification quality, to which acceptance criteria are probably the most important factor.

Besides this, Figure 7.11 and 7.12 show further negative aspects of backlog management. The consequences of conveying a false picture of agile planning, which arise from naming backlog items "issues," have already been discussed in the previous section. However, more issues within the figures are worth mentioning.

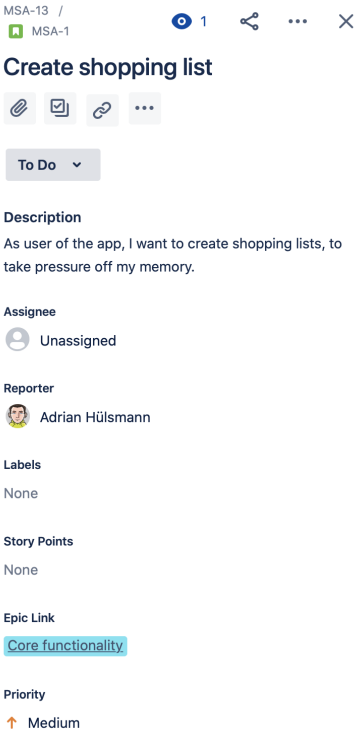


Figure 7.12: Jira issue details

The first is about backlog prioritization. At the bottom of details section, Jira provides an input field to specify the priority of an item. By default, a "medium" priority is assigned to each new item, which appears in the backlog as an orange arrow pointing up. However, as described in Chapter 4.5, it is the *order* of the backlog that is meant to always represent the priority in terms of sorting the items by their expected business value. Thus, adding any further concept of prioritization to the backlog undermines its simplistic design by adding an unnecessary orthogonal dimension. In other words, from an architectural point of view, an item by itself has no priority at all. Prioritization only comes into consideration when multiple items are in comparison with each other. Therefore, prioritization is meant to be solely part of the backlog and should be reflected by ordering items alone. Otherwise, contradictory requirement specifications cannot be avoided, for instance, when an item is specified with a "high" priority but simultaneously put to the very end of the backlog.

Another example of undermining the Scrum framework with questionable features can be seen in the possibility of assigning states to backlog items. As can be seen in Figure 7.12, the details view of a backlog item provides a dropdown to select a state, which by default is set to "To Do" and may be changed to "In Progress" or "Done." Again, this concept adds no particular value to managing the backlog or the overall Scrum framework but instead increases the likelihood of

inconsistencies in terms of sprint preparation and misunderstanding essential Scrum components. For example, what does it mean if an item of the product backlog is "In Progress" when this backlog is meant to be a collection of items for *future* development?

In Scrum, items are naturally "In Progress" once they are selected for the sprint (at which point they are transferred from the product backlog to the sprint backlog as a separate collection). Likewise, an item is "Done" only when it was implemented during the sprint and fulfills the team's definition of done, i.e., it cannot be done and simultaneously still be part of the product backlog.

However, while Jira is used as an example, problems concerning backlog management could have been identified in all of the investigated applications.

7.2.2.4 *Sprint Planning*

As described in Chapter 4.7.1, the sprint planning event consists of the following steps and therefore always has the same sequence:

- displaying the backlog to the Scrum team,
- discussing the prepared set of user stories,
- reaching a mutual understanding about the acceptance criteria,
- estimating stories with story points,
- defining the sprint backlog by choosing stories plus at least one process improvement and balancing estimation against proposed business value,
- and defining the sprint goal as a mutual commitment of what should be accomplished in the sprint.

In the following, each point is discussed and compared with the results of the studies.

Starting with *displaying the backlog* to the Scrum team and *discussing the prepared set of user stories*, it turned out that even this simple task is not as trivial as it may look at first glance and indeed not well supported by the investigated applications. The reason is that all applications provide user interfaces designed to work for a single person only but do not offer a dedicated view that suits a collaborative setting or meeting situation.

For example, during the field studies with the project groups, the backlog was displayed to the whole team by connecting the product owner's laptop to a video projector. Hence, the standard desktop view was visible to the group.

The first thing to mention is that the font sizes of the applications were too small, so user stories were difficult for some meeting attendees to read. As a result, the font size had to be increased using the standard built-in browser functionality. However, for two reasons, this was not an applicable solution. First, manually increasing the font size can lead to broken layouts. Second, this kind of "zooming in" means that relevant information is likely to no longer fit into the application's viewport, which is the visible section of the screen. Consequently, scrolling must be heavily increased to access various parts of user story information necessary for enabling group discussion. Hidden information also means higher cognitive loads because of the necessity to remember what has been seen before.

The field studies further revealed that the need to manually adapt the applications' UIs for better readability is affected by group dynamics or sociological aspects. That is because the question for increasing the font size was not posed in the first meeting. It seems that some students must feel comfortable enough to admit not being able to read what is projected on the screen. Interestingly, this was not a barrier to reach only once. Even after the groups agreed to generally increase the font size for all future meetings, the presenting person sometimes forgot it. However, the attendees did not remind the presenting person to increase the font size. Instead, it could be observed that the request was only posed after some of the user stories had already been discussed and apparently without all of the team members being able to read and understand what the stories were about.

Another downside of not having a dedicated view for the sprint planning event is that many UI elements and features are present, which are not important and, therefore, not used within the meeting. Not only is this distracting from the essential parts of what the meeting is about, but it also contributes to the problem mentioned before and further limits the available screen space.

Regarding *reaching a mutual understanding about the acceptance criteria*, it must be said that the field studies clearly revealed difficulties when acceptance criteria are not supported in the applications, as already described in Section 7.2.2.3. In these cases, the criteria were nonetheless discussed during the sprint planning. However, it was observable that the sprint outcome often derived from what had been anticipated because of misunderstandings about the product owners' explanations. In many cases, this led to unnecessary revisions of features that were implemented incorrectly.

While all of the investigated applications provided input fields for *estimating the implementation effort of a user story* with story points, features for supporting the estimation process itself show great differences.

Jira has no support for the estimation process.

IceScrum provides a slider with green, yellow, and red regions, indicating that bigger stories will likely cause more problems during implementation. Below the slider are two columns for comparing the selected estimation value with other stories (see Figure 7.13).

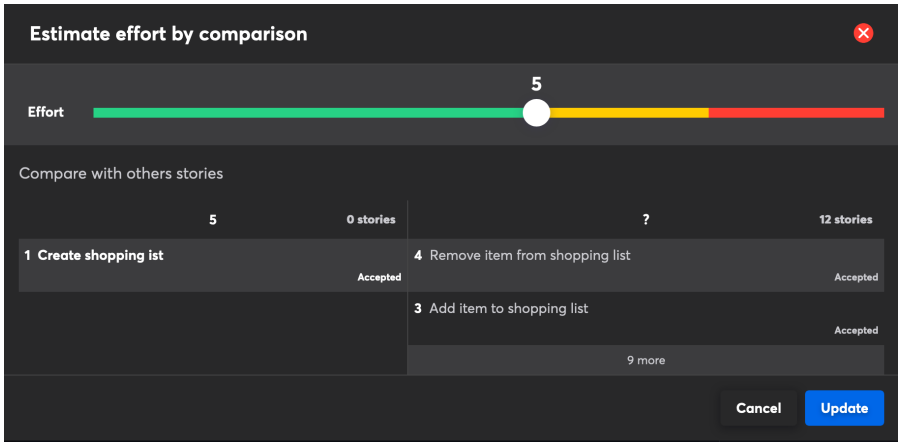


Figure 7.13: Estimation in IceScrum

ScrumDo offers an interactive planning poker session, shown in Figure 7.14. By starting the session from the backlog item's details, all other logged in team members will be notified to join. Regardless of the page the actual team member is looking at, joining the planning poker session will open an overlay, which initially shows the backlog item to vote for and a set of story points to choose from. After the user selects a value, a list on the right side will show all stories of the product backlog that have been estimated with the same value before. Like IceScrum, this serves to base the decision on references to previous estimations. Only after the own decision is locked in the other members' votes are shown, thus simulating a planning poker session with physical cards as described in Chapter 4.8.2.2.

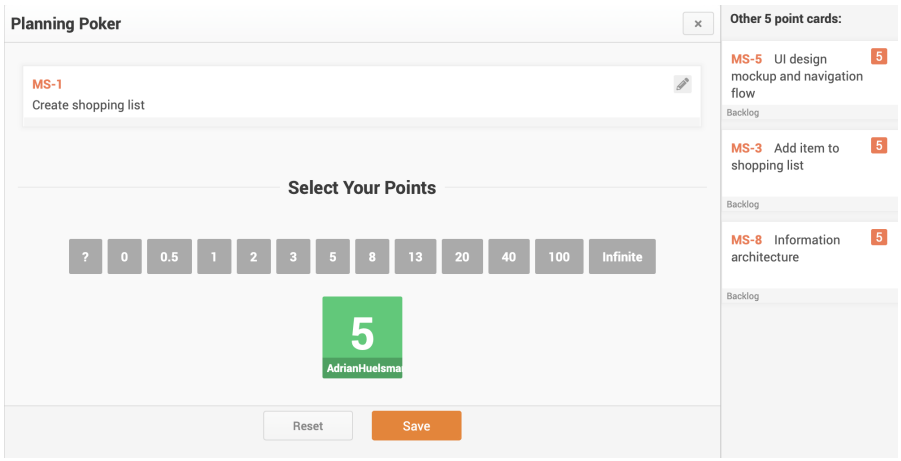


Figure 7.14: Planning Poker in ScrumDo

VersionOne supports the estimation process by a free external service called *Estimably*™, allowing teams to run a planning poker game to estimate their backlog collaboratively. When a game is started within the application, it creates a URL containing a unique ID that must be shared with the team manually. The members can then use that URL to log into the game session. Together, this procedure creates two views. The *facilitator view*, as shown in Figure 7.15, is part of the main application and shown to the person starting the session.

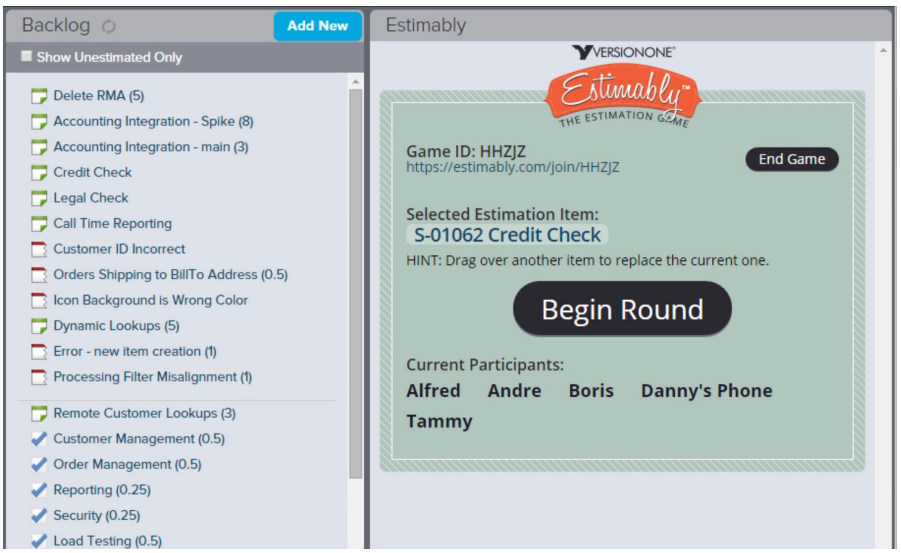


Figure 7.15: Facilitator view of Estimably

In contrast, the *participant view*, as shown in Figure 7.16, is provided through the connected external service to each of the participants who logged into the game session.

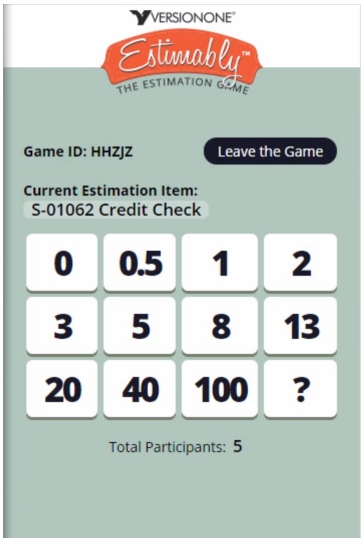


Figure 7.16: Participant view of Estimably

The initiator of the game selects an item for estimation by dragging it from the backlog into the facilitator view. There, he can see the already connected team members and start an estimation round, which then updates the participants' views so that they can choose a point value. The group result is finally revealed in the facilitator's view once each participant made a decision and took a story point vote.

During the heuristic evaluation, it turned out that all applications offering features for the estimation process have significant downsides concerning the requirement to *minimize the user's memory load*. That is because relevant story information is no longer visible once the voting is triggered so that details of the story and the acceptance criteria must be remembered. This applies not only to the details of the story to vote for but also to the details of the reference stories since they only show the title and the story point value. During the vote, it is impossible to access either detail because no links or information are embedded into the estimation views. This problem also surfaced during the field studies. By comparing sessions of analog planning poker using physical cards with the digital counterparts of the applications, it turned out that the mentioned increase in memory load did provoke greater disagreements among the team members concerning the estimation value. Thus, more rounds had to be triggered to achieve a mutual result, which slowed down the estimation process and put stress on the meeting concerning staying in the intended timebox.

Surprisingly, *defining the sprint backlog* by transferring the most valuable stories from the product backlog into a separate collection as well as *specifying a sprint goal*, also have both been more problematic than initially expected, with all except the IceScrum application.

The first issue to mention relates to violations against the *feedback heuristic*, which stem from not using state-of-the-art web technology. Modern web frameworks allow reactive user interfaces characterized by views that can automatically update to match the state of the underlying backend data. In contrast, traditional web architectures rely on synchronous data fetching and explicit server-side rendering. Thus, they may only update the page of the user who triggered the data change but cannot update the views of all other users looking at the same page. If not considered properly, this technical limitation may lead to a bad user experience, particularly affecting the flow within cooperative group settings because of resulting distractions.

Especially this became apparent when defining the sprint backlog. The applications provide convenient drag-and-drop operations to move items from the product backlog into the sprint backlog or dialogs with multiple-select options to select product backlog items, which are finally transferred to the sprint backlog with one click. However, in none of the applications is this operation reactive. That means all connected clients still show the entire product backlog, but no sprint

backlog, and people have to manually refresh the page to receive the data update. Furthermore, the implementation of drag and drop was often slow and erroneous. This technical downside particularly applies to ScrumDo since the visual feedback of the dragging interaction has so much lag that the drop-operation itself is not triggered correctly, and consequently, adding an item to the sprint often fails.

Regarding the sprint goal, the investigations revealed that although all applications offer the possibility to specify a sprint goal, they cannot benefit the Scrum team, thus making the feature itself insufficient to fulfill the underlying concept of what the sprint goal is about. What is meant by that shall be explained by the following example.

In Jira, a sprint backlog is created on the backlog page by clicking the blue "Create sprint" button (see Figure 7.17). This creates a new drop zone on top of the backlog, to which items from the product backlog can be dragged (see Figure 7.18).

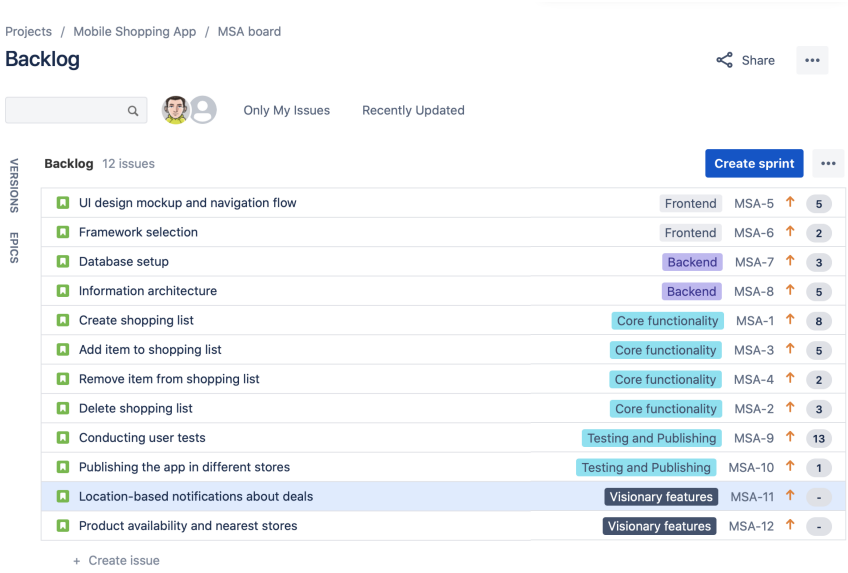


Figure 7.17: Backlog in Jira

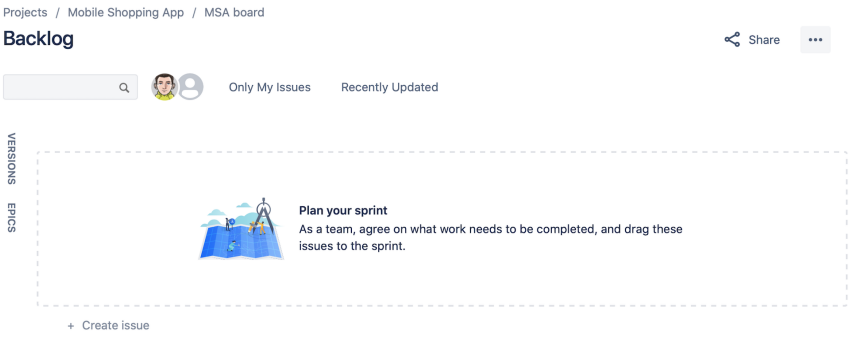


Figure 7.18: Drop zone of the empty sprint backlog in Jira

As can be seen in Figure 7.19, this new sprint backlog is given a name that is derived from the initial letters of the project name and the consecutive sprint number (in this example "MSA Sprint 1").

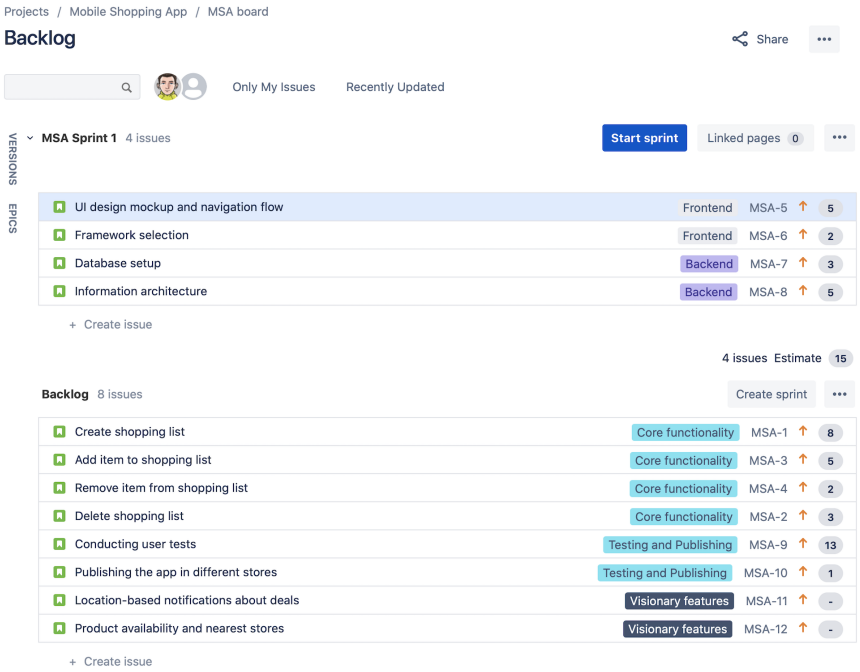


Figure 7.19: Filled sprint backlog in Jira

Once the sprint backlog is filled, the sprint can be started by clicking the "Start sprint" button, which opens the dialog shown in Figure 7.20.

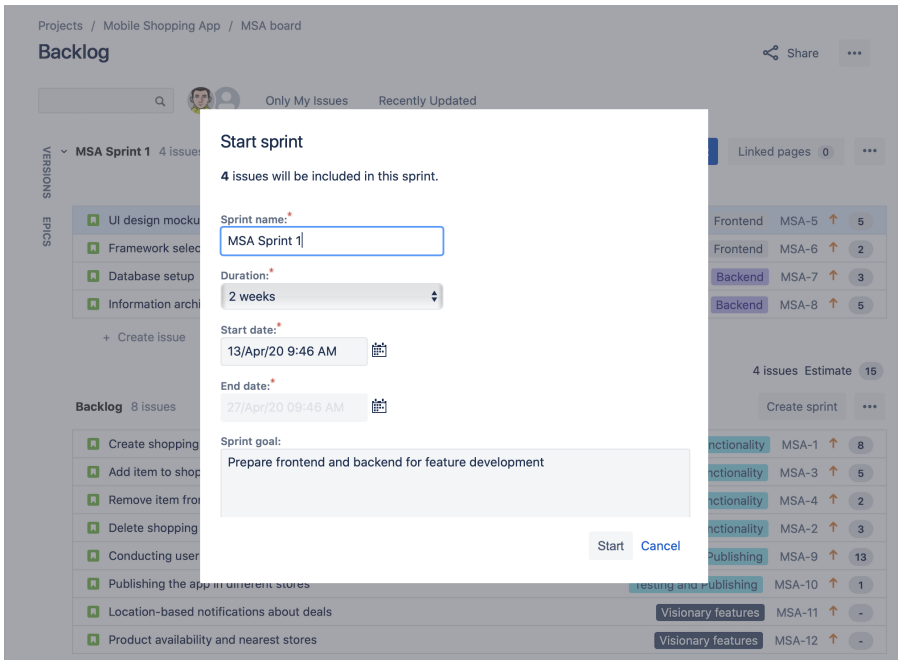


Figure 7.20: "Start sprint" dialog in Jira

Here, the team is able to change the default name of the sprint, define the sprint length, and specify the goal. However, the important point is that the sprint planning event is not goal-driven when the goal is specified at the *end* of the planning process. Instead, it should be specified at the very beginning because, as Rubin says, the sprint goal "describes the business purpose and value of the sprint," and by providing a "clear, single focus" for the sprint planning meeting, it becomes "the foundation of a mutual commitment made by the team and the product owner" [226, p. 69].

That is also why the goal should always be present to the whole team during the entire sprint because it creates awareness of what to achieve, and "by adhering to a sprint goal, the Scrum team is able to stay focused [...] on a well-defined, valuable target" [226, p. 69].

However, even if input fields are provided, all of the investigated applications consider the sprint goal not more than being of secondary importance because one half is either not displaying it on other pages at all (IceScrum, ScrumDo) and the other half is using designs in which the sprint goal is upstaged by other interface elements (Jira, VersionOne), thus making it unable to increase the team's awareness, as shown in Figure 7.21.

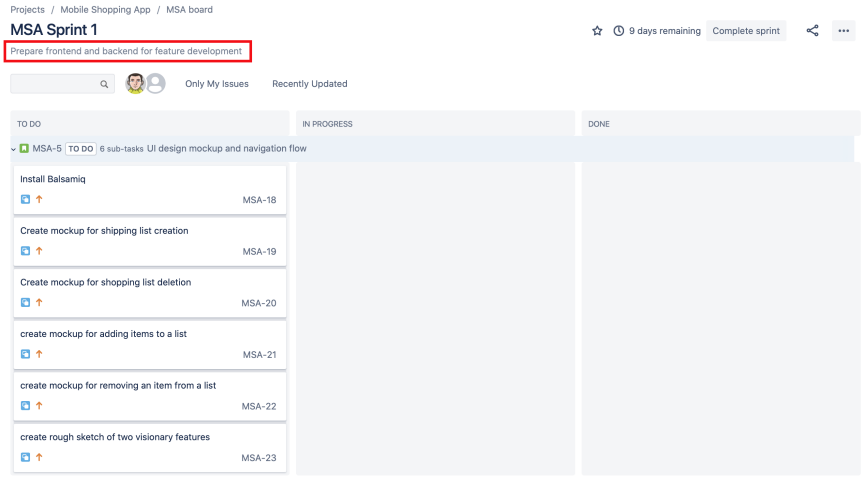


Figure 7.21: Small display of the sprint goal in Jira (highlighted)

7.2.2.5 *Sprinting*

After the sprint planning, the development team starts the implementation work and turns the requirements of the sprint backlog into working code. During this sprinting, developers use several other applications, like IDEs, test suites, and code repositories, which provide functionality for creating the product increment in the form of executable and potentially shippable source code.

While many agile ALM applications provide integrations to some of these developer tools, their primary purpose in supporting the implementation process lies in the *management of development tasks*.

Overall, all of the investigated applications support this well. Developers can easily create tasks, assign team members, quickly access their personal tasks, and update the status according to their implementation progress. However, three main problem areas have been identified.

The first problem relates to the already mentioned *non-support of acceptance criteria*. Since the underlying idea is to establish concrete testable conditions for validating that the implementation meets the expectations of the product owner, acceptance criteria also build the foundation for the development tasks. Hence, no matter how well the management of tasks is supported, as long as the development team does not know how to validate the requirements, fundamental misconceptions are very likely.

The second problem concerning *missing reactivity* due to technical limitations of the applications has also been mentioned before. Regarding sprinting and managing tasks, the field studies revealed that this leads to a severe drawback regarding Scrum's demand for transparency (see Section 4.1). Because of missing reactivity, team members often have not been informed about important updates. For instance, the update of a developer picking a task could not be automatically pushed to the connected clients of the other members. Consequently, their user interfaces were outdated until they would manually refresh the page to receive the data change. However, manually refreshing the page to receive data that may eventually be changed by others is not what people expect. Therefore, it is not surprising that the studies revealed that although people were aware of this technical limitation, they nonetheless forgot to update their views and frequently provoked data conflicts.

Depending on the actual context, it turned out that missing reactivity may have severe consequences. For example, several times, two developers picked the same task without knowing it and thus unnecessarily duplicated the work. Another example was when team members did not automatically receive information about arising impediments blocking certain user stories or preventing tasks from being started. In these cases, for instance, when the impediment lies in ambiguities that must be eliminated by the product owner at first, it happened that developers, being unaware of these impediments, nonetheless started to work but later came to realize that the requirements have changed in the meantime so that their work became useless. Not only is this demotivating for the team, but it also endangers the sprint goal, especially if the team does not consider enough buffer time during the sprint planning.

Lastly, none of the investigated applications allow the development team to check their implementation work against a *definition of done* (see Section 4.3). In this respect, the only application that at least provides some functionality is IceScrum, which offers an empty text field for every sprint to specify a definition of done. However, the problem with this is that this definition is not copied or stored on a higher level but must be re-entered for every sprint. More importantly, it has no connection to items of the sprint backlog. This means that the "done" state of user stories cannot be assessed and verified against the individual criteria of that definition because it is specified as free-text (and not as a list of checkable criteria), and even worse, it is not shown on the story's details page. This issue must be considered crucial since it was observable in the field studies that not using a definition of done diminishes the overall code quality of the delivered product, which matches the relation between technical debt and the definition of done given by Rubin:

> *"Work that we should have performed when a feature was built, but ended up deferring until a later time, is an important cause of technical debt. Using Scrum, we want a strong definition of done [...] to help guide the team to a low- or no-debt solution at the end of each sprint."* [226]

In the course of several sprints, neglecting a definition of done leads to an increased amount of refactoring, which in turn limits the overall return on investment. The field studies teams, for example, both came to the point where they needed to insert "refactoring sprints." These are sprints with the only goal of performing technical debt reduction and code optimization work. According to Rubin, such sprints "are to be avoided whenever possible" [226, p. 159] because, during this period, the team is not going to deliver any customer value. For that reason, using a definition of done ensures raising the team's awareness that reducing technical debt is something to deal with a little bit each sprint so that refactoring occurs incrementally instead of using a sledgehammer method.

7.2.2.6 *Daily Scrum*

As explained in Section 4.7.2, the idea of the daily Scrum meeting is to synchronize the work between all development team members and provide transparency about the progress towards fulfilling the sprint goal. It is usually held as a standup meeting to underline the importance of its fifteen-minute time limitation. This timebox should be sufficient to update the team about what has been done since the last daily Scrum and what is intended for the current day. It should also allow clarification of any arising impediments so that other developers or the Scrum master can offer their help.

Based on the field and feature studies, it must be said that the applications investigated do not support this process well. None provides dedicated views for co-located group scenarios or any particular assistance for the daily Scrum meeting.

In particular, this means that the user interfaces are not designed to be viewed by multiple persons in co-located work settings, e.g., by displaying the applications on a projector or (as in the case of the project groups' daily meetings) on a large screen. This is because small font sizes are hard to read from some feet away, leading to the problems already discussed in the sprint planning event (see Section 7.2.2.4). Apart from this, the applications also have no particular functionality to support the daily Scrum event in a *meaningful* way.

For instance, due to the lack of alternatives, both project groups of the field studies decided to use the applications' *task boards* for their daily Scrum meetings. While this view generally provides a good overview of the actual work status, and although some of the applications also allow filtering the tasks by single team members, there is still one major drawback: none of the applications is designed to cover any meeting support, e.g., information like the burndown graph is spread through various pages and not accessible on a single page. This has already been discussed in terms of inadequate views and graphical interfaces, but here, it must be mentioned that this limitation is on a much more structural prospect with regard to the applications' architecture. Specifically, this means that meeting-specific data is not stored. Consequently, it is impossible to relate to other daily meetings that were held previously.

Hence, when members of the project groups wanted to answer what they did since the last daily Scrum and what they intend to work on for the current day, it caused them a lot of *memory stress*. That is because they needed to *remember* what has been prospected in the last daily meeting in order to compare it against what has been achieved. However, this comparison is an essential aspect of the daily Scrum since differences provide hints about possible impediments to the rest of the team, which may be unconscious to the person speaking.

From Chapter 4.7.2, it should be recalled that transparency about the difference between the previous and current daily Scrum is the foundation for continuous inspection of the development progress, which in turn is the foundation for possible adaptions to ensure that the team will meet the sprint goal. For that reason, the limitations of the investigated applications can be considered significant.

This is not only because of the team members' memory stress but also due to the observable negative effects. From the experience of both project groups, it could be seen that the need to remember disturbs the overall flow of the daily meeting and thus not only increases the

risk for people not listening to others (see Section 6.2.6.3), but also provokes the unwillingness to provide help by other team members (see Section 6.2.6.5), since impediments would not have been broad to light.

7.2.2.7 *Grooming the Backlog*

While Section 7.2.2.3 already explained problems regarding the general management of the backlog, this section focuses on grooming it, which differs in that it is a *collaborative* effort between the product owner and some members of the development team, whereas general backlog management is done by the product owner alone.

As written in Section 4.5, *grooming* describes the action of refining the backlog as preparation for the next sprint planning event. This includes presenting the prepared backlog to some developers, who may then ask questions that typically would arise during the planning event. However, by asking these questions earlier, the product owner is given a chance to act on unsolved issues and better prepare for the actual planning event to avoid problems that might arise and that might prevent an important item from being included in the sprint.

For that reason, to have sufficient time to address the identified questions, Cohn suggests scheduling this short meeting three days before the end of the current sprint [60]. Though it is the responsibility of the product owner, he or she is not the only one who benefits from this action since grooming contributes to establishing a shared understanding of the product vision within the whole Scrum team and aligns members in terms of the work left to be done [249].

However, as Padmini et al. found out, many teams do not consider the backlog grooming activity during their projects. In their studies, only half of the respondents mentioned conducting this ceremony. While the authors think that "this indicates that either scrum teams do not know about the importance of the [...] backlog grooming ceremony or they are not considering it as adding value to the project," they also mention that those respondents were the ones with "no idea about the story acceptance criteria and definition of done," as well as the ones without "properly defined stories in their projects" [201].

Therefore, collaborative backlog grooming activities are, in fact, an essential part of Scrum. However, it turned out that the tools investigated did not support this ceremony directly. While the backlog can be viewed and edited by a single person working at a personal computer, none of the investigated applications was designed to support a *collaborative* way of working, e.g., when the computer is connected to a projector as a result of the usability problems that have already been mentioned in Section 7.2.2.4.

Besides this, it was impossible to display multiple elements of the backlog so that their details would be visible simultaneously. However, this would have been a benefit because members of both project groups independently reported having difficulties with comparing various story contents during the grooming activity when the individual details were not accessible at the same time. This made the grooming activity unnecessarily complex.

In addition, using a projector and standing in front of it meant that things discussed were often not recorded in writing. This was because the switch to typing with the connected keyboard led to pauses in the discussion and thus disrupted the overall flow of the collaborative ceremony. As a result, it was often decided against using the keyboard for taking notes, i.e., notes were not taken at all. However, in many cases, this led to the need for renewed queries because things that had been discussed were sometimes forgotten after the session had been completed.

7.2.2.8 *Review Meeting*

As explained in Chapter 4.7.3, the main objective of the sprint review is to inspect the outcome of the current sprint in order to adapt to future development. The main focus here is on *feedback* from stakeholders and customers, which significantly influences what will be implemented in the subsequent sprint. However, as explained in Chapter 6.2.7, common problems of the sprint review include passive and unengaged invited participants (see Section 6.2.7.1), losing focus on relevant content (see Section 6.2.7.3) and treating the review as venue of approval (see Section 6.2.7.4).

These problems were confirmed in the field studies and occurred equally in both project groups. In the feature analysis of existing project management applications, it also became apparent that no vendor provides explicit functionality to support the sprint review for the actual meeting or to prepare for it during the sprint. This has serious consequences, which are explained in the following.

As Rubin makes clear, the inputs to the sprint review are the sprint goal, the sprint backlog, and the sprint outcome, which is the potentially shippable product increment [226, p. 367]. While none of the investigated applications offers dedicated views for the sprint review, these basic elements are nonetheless accessible in the provided task boards (except for IceScrum and ScrumDo, which do not display the sprint goal at all). Hence, all reviews of the field studies were conducted using the applications' task board views, which corresponds to the usual procedure in the industry, as was later confirmed by the interviews.

However, the task board view is problematic and may induce some of the aforementioned review problems. This is because of the following reasons.

1. Tasks are not relevant and lead to discussions.
2. There is no clear distinction between finished/unfinished stories.
3. The reasons for failure are not visible.

The first thing that could be observed was that the visibility of development tasks leads to unnecessary discussions in the sprint review. However, these tasks primarily serve the development team. They are not relevant for the product owner or the customer because the common level of understanding between all parties is *user stories*, which are the basis of all planning activities. The visibility of tasks shifts the basis for discussion to a deeper level, which is not only more fine-grained but also often technical, covering implementation details that are not relevant for obtaining feedback on the sprint result. Special care should be taken here because by shifting the discussion level to technical details, invited guests can be quickly disconnected, which can make them behave rather passively and reluctantly, leading to the problems described in Section 6.2.7.1.

The visibility of tasks also contributes to another problem. This is because the provided view of the task board seems to weaken a clear distinction between finished or unfinished items of the sprint backlog. For example, it could be observed that user stories were more often declared as "finished" and thus discussed in the review, although, in fact, they were not. This was especially the case in situations where most of the tasks had already been completed and, therefore, could be seen on the right side of the board. Depending on the relation between finished and unfinished tasks and their position on the board, the threshold to consider the corresponding story as finished seems to get lower or higher. Less unfinished tasks seem to increase the likelihood of accepting the story for the review, whereas more unfinished tasks seem to increase the probability that the story is not treated as being "done." More precise statements about this relationship cannot be made and would require further investigation. However, the important point is that the visibility of tasks seems to influence the decision of whether a story is finished at all. Not only does this weaken the "definition of done," which can lead to software quality issues, but it also causes the decision on acceptance to be moved to the review meeting itself, thus leading to treating the review as the venue of approval with the corresponding problems discussed in Section 6.2.7.4.

Lastly, the field studies and feature analyses revealed that the task board is not suited for conducting the review meeting because it does not provide an opportunity to present comments about problems

that occurred during the sprint. Although the primary purpose of the sprint review is to generate feedback on the completed features, it is often forgotten that short explanations at the beginning about why the other items could not be completed are also a valuable part of the meeting and in line with the requirement for inspection and transparency. Therefore, a brief look at the requirements that have not been achieved is useful in two respects. Firstly, to generate possible insights for the next sprint and secondly, to mentally check off the unfinished features and thus concentrate on reviewing the product increment.

This second point, in particular, was confirmed positively in the field studies. This is because both of the teams examined found in the course of their projects that their sprint review meetings often led to discussions about the items that had not been implemented. In the subsequent retrospective, it became apparent that these meetings were sometimes assessed as strenuous and not productive, which meant that the teams had to take action and decided that unfinished items should be addressed at the beginning of the review in order to be able to devote themselves entirely to demonstrating the sprint outcome. During later process analyses, both groups assessed this procedure as very useful and maintained it until the end of their projects. During the review meeting, however, the participants had to gather background information on unfinished stories, as these were often hidden in the comments on the respective story pages, and these comments were not prepared or made available on the task board. This resulted in frequently switching between the task board and the detailed views of individual stories, which was perceived as distracting, so one project group decided to write down notes about problems in the sprint on an extra sheet of paper to use it during the review.

After the inspection of the sprint backlog, the adaptation part of the sprint review follows, i.e., the feedback gained on the increment flows directly into the planning of the next iteration. This is done by revising and reprioritizing the product backlog, which may also result in changes in the release plan [226, p. 368f]. In the end, the outcome of the sprint review is a clear high-level roadmap for the further development process, whereby details of individual stories (such as the exact elaboration of the acceptance criteria) are usually worked out by the product owner until the next sprint planning event.

For this adaptation part, the demand for tool support lies in quick and easy changes between the view, where the sprint feedback is actually gained, and the product backlog, where this feedback is then considered by revising and reprioritizing new and older requirement specifications. However, it turned out that this change between views may mean several clicks because of complex menu layouts and navigational architecture, which become frustrating since the switch between

product backlog and task board is not done once but multiple times during feedback processing. This would have been a minor problem if the user could use the browser's back and forward navigation buttons. However, this workaround was not a solution for three of the four applications because they did not correctly preserve the application state, i.e., the view of the last page visited.

Overall, there is a higher cognitive load when the product backlog is not immediately accessible, which disrupts the flow of the meeting. As a result of this tool limitation, during the field studies, feedback collected during the sprint review has not been worked into a revisioned backlog immediately. Instead, the product owner (myself) decided to take notes on paper. While this preserved the feedback flow, it disconnected the other attendees from planning the future iteration, making it difficult to establish a shared picture since the notes were not visible to all team members.

7.2.2.9 *Retrospective Meeting*

While the sprint review is the central learning loop for assuring that the team is building the most valuable product, the sprint retrospective is the fundamental security mechanism for ensuring that the team stays in a healthy and satisfied condition by reflecting its own work practices and identifying improvements for the development process. Hence, it is probably the most important of all the Scrum meetings. It is where all team members consciously take their time to share thoughts, openly speak about new ideas, discuss existing problems, and mutually decide how to improve as a team.

Surprisingly, the studies revealed that only one of the investigated applications provides basic features for a retrospective meeting, whereas support is practically nonexistent in all others.

In **Jira**, completing a sprint will automatically open a sprint report showing a burndown graph and a popup asking the user to create a new retrospective page, which is nothing more than a link to *Confluence*, an external wiki application of the same tool vendor. This means, on its own, Jira offers no features for conducting a retrospective meeting, and even by taking into account the Confluence application, support is limited to manually creating shared text documents.

The same holds for **IceScrum**. Although no external services are necessary here, the support of the retrospective again only consists of an input option for text. Moreover, Figure 7.22 shows that this textual description is stored on a global level in the dashboard for the whole project. This means, as opposed to Jira and its Confluence integration, there is no connection to individual sprints or teams, which further limits the usefulness of this function.

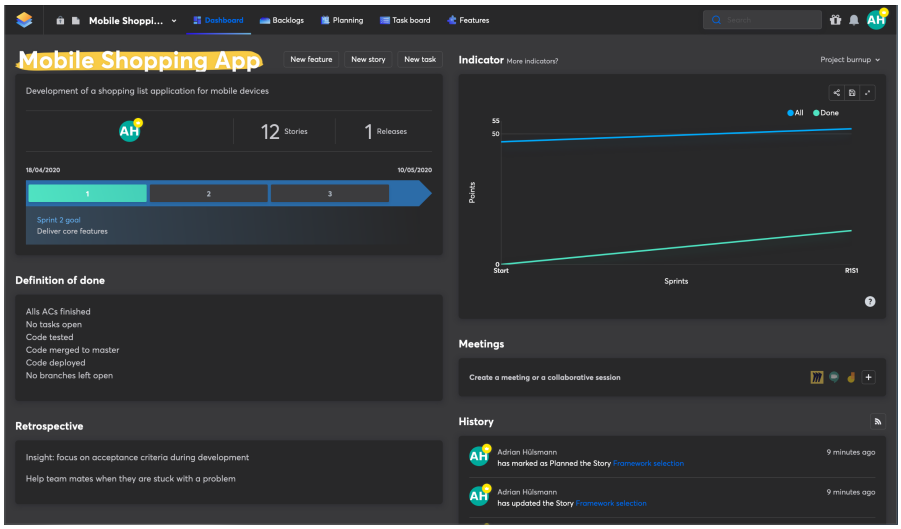


Figure 7.22: "Retrospective" text field in IceScrum

ScrumDo has even less support and neither provides any input option nor mentions the sprint retrospective.

In **VersionOne**, functionality for conducting a sprint retrospective can be found in a sub-menu-group, which is labeled as "Review" and is organized under the top-level menu entry "Team" of the main navigation (see Figure 7.23).

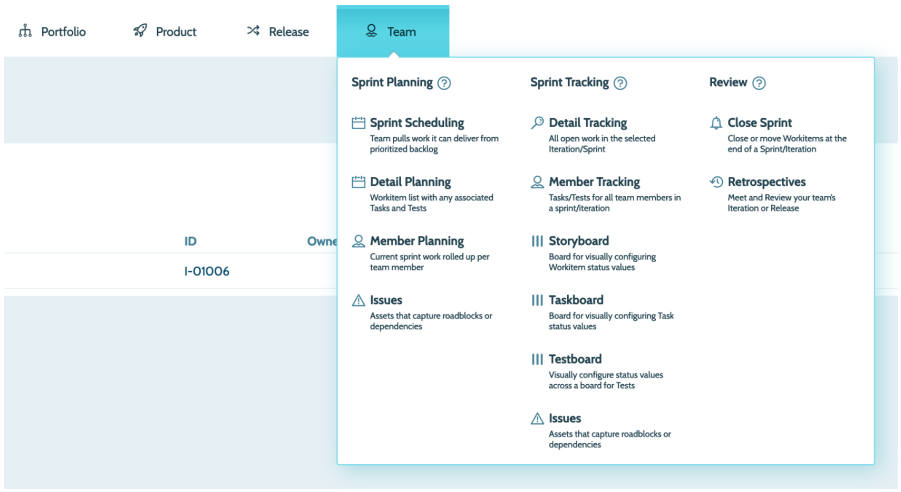


Figure 7.23: Top-level menu "Team" in VersionOne

This menu is interesting because it tries to organize features along the natural flow of a sprint, starting with "Sprint Planning" on the left side, over "Sprint Tracking" in the middle to "Review" on the right side. While the concept of mirroring the course of a sprint with the navigational architecture of the application is generally a good idea because it is likely to help users find the right feature for a given situation, VersionOne is not capable of doing so. The dialog breaks with the usability heuristic "simple and natural dialogue" because of

confusing menu item allocations, which becomes evident since the entry "Issues" appears not in one but two of the navigational groups. In addition, it also breaks with the heuristic "consistency" by assigning "Retrospectives" to the menu group "Review," which is irritating since these are special Scrum terms meaning different things. A further issue with this menu is that the entry "Retrospectives" is below an entry for closing the current sprint, again suggesting some ordering along the course of actions. However, closing the sprint before conducting a retrospective is causing problems, as will be explained shortly.

Although it is not immediately apparent due to the single menu entry, the retrospective feature is actually separated into *two* steps. One is for *preparing*, and one is for *conducting* the meeting. Preparing a retrospective is done by clicking the "Retrospectives" menu entry, which opens the page shown in Figure 7.24, where all currently prepared retrospectives are listed.

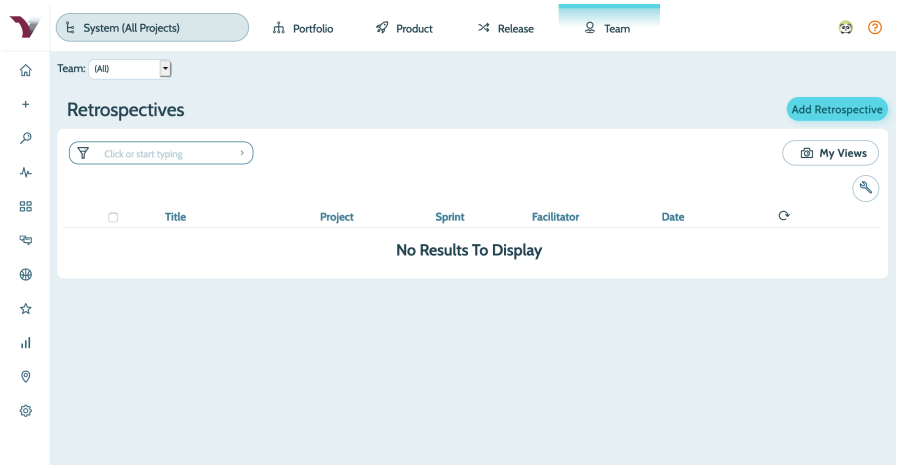


Figure 7.24: Retrospectives list in VersionOne

Clicking the "Add Retrospective" button opens the dialog shown in Figure 7.25, where the user is provided with options to specify a title and assign the retrospective to a project, sprint, and team. For the assignment of a sprint, it is necessary that the sprint has not been closed already. However, this will likely happen because of the aforementioned menu structure and interaction flow.

From a usability perspective, this violation of the heuristic "prevent errors" is highly problematic because stumbling blocks in planning a retrospective could prevent it from being held in the first place. Moreover, from a technical perspective, the manual assignments to a project, sprint, and team could be avoided if the application is built upon an architecture according to the rules of Scrum. What is meant by that is that teams are naturally associated with a sprint (and project) because of a one-to-one relation. Hence, the requirement to assign both makes the preparation for the retrospective unnecessarily complex.

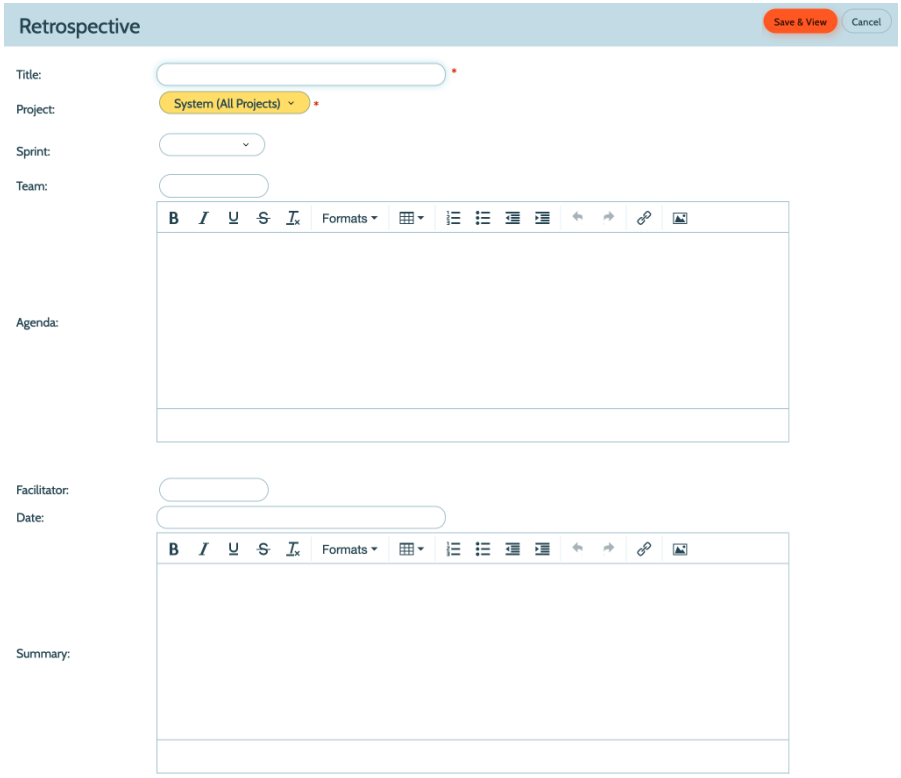


Figure 7.25: Preparing a retrospective in VersionOne

Apart from these assignment options, the retrospective preparation in Figure 7.25 offers two text inputs. The first allows setting up an agenda, which may be helpful to introduce the topics to discuss to the actual meeting participants. This helpfulness is especially true for invited people who are not part of the core Scrum team and do not participate on a regular basis.

The second text field is for creating a summary. However, it remains unclear whether this is meant as a summary of the meeting itself (which would not make much sense since, at this point, the meeting has not been conducted yet) or as an option to specify the summary of the sprint. This, in turn, would not be useful either because the retrospective is about mutually inspecting and analyzing the sprint data in all of its facets, which should not be summarized in advance (especially not by a single person) to preserve an unaltered starting point.

After the preparations have been saved, a new meeting shows up in the retrospectives list, as shown in Figure 7.26.

The prepared retrospective can then be started by the "Conduct" button, which opens the meeting view that is shown in Figure 7.27.

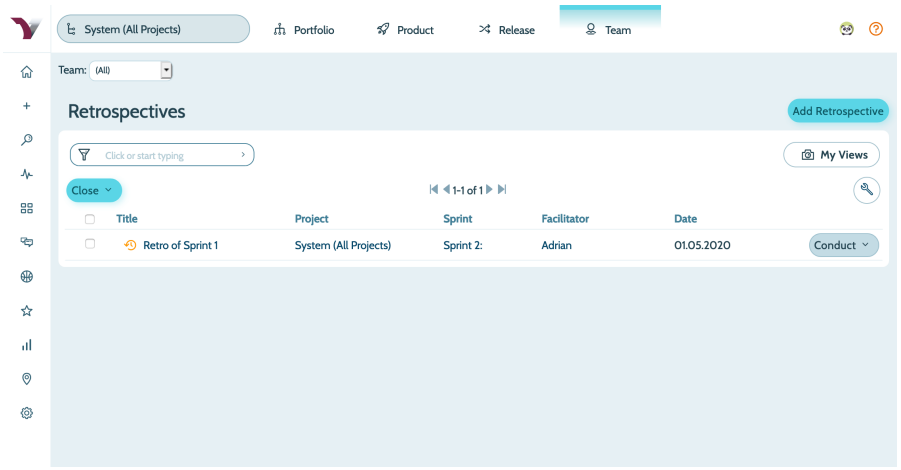


Figure 7.26: A prepared retrospective in VersionOne

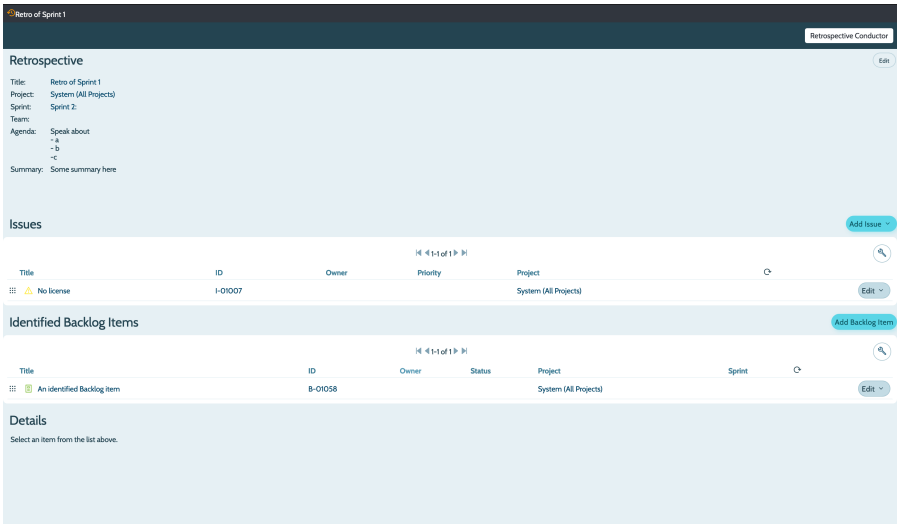


Figure 7.27: Meeting view of a retrospective in VersionOne

As mentioned before, the meeting views in VersionOne are unsuited for group settings, where the page is displayed on a projector. As seen in Figure 7.27, the font sizes are too small to read even from a small distance. Besides that, the possibility of conducting a retrospective is also highly limited. Apart from displaying the agenda and summary of the retrospective preparation, the meeting view consists of two lists only. The first labeled as "Issues" is for collecting problems as a result from a mutual sprint inspection, while the second labeled as "Identified Backlog Items" is for creating process improvements.

A fundamental limitation of the first list lies in the lack of support for actually getting to the point of *identifying* issues. Neither is any sprint data shown from which the team could derive insights and identify problems within their development process, nor are there any links to possible data sources the team could inspect. For instance, the task board is not accessible during the retrospective because VersionOne

hides the main navigation during the mode of conducting a meeting. What is more, once the retrospective meeting has been closed, it is automatically deleted and no longer shows in the retrospectives list of Figure 7.26. Hence, there is no potential workaround to access any sprint data during the meeting. Overall, not having access to the sprint data during the retrospective is a massive violation of the heuristic to "minimize the user's memory load" and limits the possibility of identifying problems.

Regarding the second list, the term "backlog items" violates the heuristic "consistency." It further breaks with the whole retrospective concept because, at the end of the meeting, the team should agree on improvements (sometimes called insights). These are not "backlog items" in the proper sense because this term is usually a shortened form of "product backlog items." However, improvements do not relate to the product itself; they relate to a given team and describe actions to enhance the development process. Since the feature to specify improvements is not given in VersionOne, the usefulness of the whole meeting functionality for retrospective purposes must be strongly doubted.

7.3 CONCLUSION

This chapter aimed to derive a status quo of Scrum tool support by investigating a selection of agile project management applications and analyzing their limitations regarding usability, features, and functionality. The purpose of this section is to conclude the status quo from the many individual findings presented before.

In summary, the identified problems can be categorized into three areas, which are:

1. general usability issues,
2. missing out core elements of the Scrum framework,
3. and limitations in cooperative work support.

General usability issues

General usability issues may range from only cosmetic effects to very severe problems, all of which have been identified in any of the investigated applications. One of the minor issues is, for example, that the user is shown the internal ID of a backlog item or task, which does not correspond to his natural language and does not offer any added value. However, using this internal ID can quickly become a major problem if it is not only displayed but also used for reference, e.g., in a search field, and the user is forced to remember these cryptic strings of characters and numbers.

Technical weaknesses

Furthermore, usability problems may result not only from design deficiencies but also *technological weaknesses*, leading to bad interactive experiences, as in the case of Jira and ScrumDo, in particular. These include, for example, the fact that changes to the database are not automatically reflected in an update of the user interface. Thus, a user may not notice changes made by others unless he or she has actively reloaded the page beforehand. However, field studies have shown that this reloading is often forgotten, even when users are aware of the technological limitations of an application. This is because today's web applications quickly give the impression of native applications due to their structure and visual appearance, and thus, corresponding feedback is also expected from the ground up. Therefore, weaknesses in the technology can easily lead to frustration and confusion.

High complexity

However, the negative usability also results from the *high complexity* of the applications examined. This is reflected, among other things, in nested menu structures or a large number of settings or features that are not likely to be relevant for most users.

Low consistency and missing feedback

Moreover, *low consistency and missing feedback* of the applications also lead to an unnecessary additional load for the users. For example, even when predefined templates for Scrum are selected, terminologies other than those of the framework are sometimes used in the applications. In addition, system states are possible (or in some cases even necessary) that contradict the rules of the Scrum framework. This includes multiple sprints of a team taking place in parallel, which is against the core concept of sprinting, or setting a processing state of a backlog item in terms of its implementation, as described at the end of Section 7.2.2.3.

Missing features

In addition to usability deficiencies, which, as already mentioned, can range from merely minor effects to major problems with regard to tool usage, it is above all the *missing features* that can have severe consequences for the entire way of working in accordance with Scrum.

No guidance

It is noticeable that the applications examined have *no guidance* through the clearly defined Scrum process or consider it in the user interfaces. Except for IceScrum, which at least offers a clearly defined sequence of actions in the context of backlog management up to the planning of a sprint, functions are scattered and do not follow a clear structure, which the scheme of a sprint would specify.

Inadequate meeting support

In addition, it is noticeable that the focus of the applications examined is primarily on the planning aspects, whereas significantly less attention is paid to the actual activities within a sprint that build on this. For example, there are neither functions for a sprint review nor for conducting a sprint retrospective, which also leaves out inspection and adaptation as essential cornerstones of empirical process control (see Chapter 4.1). Therefore, in practical use, other tools must

be used additionally, which, on the other hand, can lead to media discontinuities, synchronization problems, and, therefore, to a higher expenditure of time. In the case of the project groups, these additional tools were mainly notes and pens to track the product or process changes identified within the review and retrospective meetings.

Weaknesses of central Scrum components

In addition to the *inadequate Scrum meeting support*, other *weaknesses of central Scrum components* were also revealed. For example, the role model is not mapped in any applications examined or addressed by a particular range of functions for developers, product owners, or Scrum masters. Established measures for increasing product quality, such as acceptance criteria, the definition of ready, and the definition of done, are also insufficiently taken into account.

Limited support for collaborative activities

Lastly, the identified problems reveal strongly *limited support for collaborative activities*. This is due to the fact that all of the applications examined are intended to run on desktop computers and were thus designed to be used by a single person. The resulting problems manifest most notably during the Scrum ceremonies. For example, if the backlog is displayed on a beamer during sprint planning, the content is difficult to read even from a short distance. Within the project groups, it was observed that these deficiencies in the display could already have such an effect that people become less involved during a meeting due to the poor legibility of the information displayed.

Another example is the hurdles concerning the daily Scrum meetings. Although all applications can display the overall progress of a sprint in the form of a task board, in the meeting, it is primarily a matter of each developer briefly communicating his own individual progress so that further tasks and any arising obstacles become transparent to all team members. For this purpose, it would be helpful, for example, to display the changes that have occurred since the last daily meeting *automatically* so that people do not have to remember them laboriously.

In summary, all of the meetings provided for Scrum are only insufficiently supported. In the case of the review and the retrospective, the applications do not even offer any dedicated functions. However, it is precisely the meetings and their processes that create the framework for a collaborative working method in Scrum, which is oriented towards the principles of empirical process control. Here, the inspection of a work genesis and the adaptation of the working method must be carried out jointly. Review and retrospective are the "most important learning loops" [226, p. 364], and ensure that the entire team participates in the ongoing learning process.

In addition to the lack of features, a possible reason for the insufficient meeting support could be the applications' implementation and design as classic desktop applications representing single-user

interfaces. The mouse and keyboard interactions and the underlying paradigm of the single point of focus limit the applications' usability to scenarios with a single-person operation. Therefore, such applications reach their limits in collaborative work environments involving several people. This could also be observed within the project groups, where the dynamic interaction sequences in meeting situations led to media discontinuities and second-intent disruptions, which negatively affected the meeting flow.

Thus, in addition to developing missing features and optimizing the identified usability issues, there is also the question of how collaborative Scrum meetings could be better supported by other technologies and input paradigms.

Therefore, the next chapter considers *Natural User Interfaces* (NUIs) as successors to classic desktop applications before concluding the thesis with a novel Scrum management application proposal, particularly addressing the identified issues.

Part III

THE IMPLEMENTED SOLUTION

Up to this point, this thesis explored the challenges and issues when implementing the Scrum framework, and it identified the limitations of current project management applications to help Scrum teams face these problems.

While the investigations uncovered a significant gap in supporting key elements of the Scrum framework, particularly collaborative activities represented by the obligatory meetings, this brings into question whether traditional, single-user interfaces, such as laptops or desktop computers, are sufficient for fostering the particular demands of agile Scrum teams.

As a result, Part III explores *Natural User Interfaces* (NUIs), which are known for their potential to support collaboration, especially in face-to-face interactions.

By investigating the third research question, "What could a novel Scrum tool look like that utilizes NUI technologies for collaborative activities?", this thesis finally proposes an "interactive Scrum space," whose objective is to combine traditional single-user interfaces with various touch-based NUIs to offset their individual shortcomings, thus creating a more effective and collaborative work environment.

8

NATURAL USER INTERFACES IN AGILE ENVIRONMENTS

In contrast to other interaction paradigms, the term "Natural User Interface" (NUI) is not as simple to grasp. This is because of the question of which aspects make an interface "natural" and what is meant by that. To provide more context when answering this question, it helps to briefly look at how NUIs have evolved from the evolution of human-computer interaction paradigms.

8.1 NUIS AS AN OUTCOME OF HCI EVOLUTION

Historically, human-computer interaction began with the paradigm of *physical operation*. Early computing machines that could fill an entire room consisted of tens of thousands of vacuum tubes, capacitors, resistors, and relays, all contained and grouped within various control and calculation units. Programming these machines meant plugging cables and physically connecting the various units through electrical wires before using punch cards for both system input and output. However, since "interaction" describes an *interplay* of at least two actors through a shared communication channel, it may be questioned whether "physical operation" can be considered the first interaction paradigm due to the missing bi-directional input-response cycle.

Even if working with computers in the early days only meant operating and "acting" by feeding the system with information without immediately processing the system output, it is nonetheless mentioned here to provide a more complete picture. As shown by Figure 8.1, the evolution of "real" interaction between humans and computers can be divided into three main paradigms.

Figure 8.1: HCI paradigms[1]

1 Source: https://en.wikipedia.org/wiki/Natural_user_interface

Command-line interfaces

The first is *command-line interfaces* (CLI), which came to arise during the mid-1960s in the form of *terminals* and *consoles*. In CLIs, the shared communication channel between user and machine is based on textual in- and output and the interaction follows a sequentially strict input-response cycle. It begins with the user entering specific *commands* and associated parameters as character strings via the keyboard. The system then executes the command by an interpreter known as *shell* before the operation result is printed on the screen. To indicate the state of the interaction cycle, CLIs use a *prompt*, which either signals that the system is ready and waiting for input or otherwise not available during the execution of a command. Although CLIs are the oldest form of human-computer interaction, they are still very common. That is because commands and associated parameters represent a direct and, hence, very fast way of telling a machine what to do. With later improvements, such as multi-line commands, command blocks, and endless possibilities for shortcuts and granular control, CLIs are still indispensable for many expert environments like server environments and remain the foundation of all Unix-based operating systems, as shown in Figure 8.2.

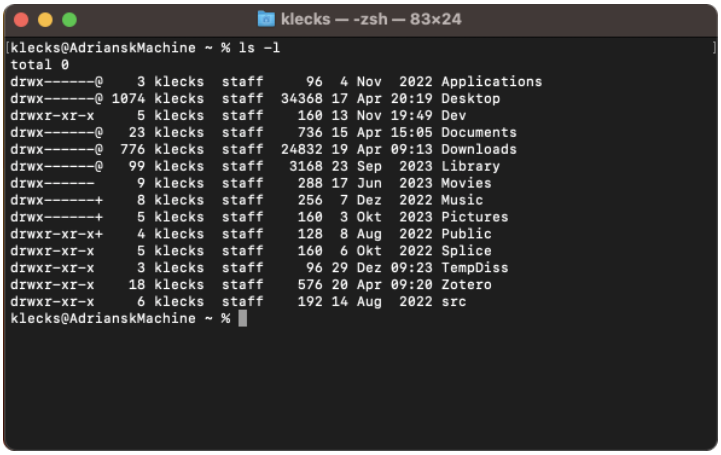


Figure 8.2: Command-line interface

Graphical User Interfaces

The first paradigm shift in human-computer interaction describes the change from CLIs to *graphical user interfaces* (GUIs). This shift was already heralded in 1968 at the Stanford Research Institute, when *Douglas Engelbart* presented a futuristic computer system intended for "augmenting the human intellect" [81].

In the now famous "Mother of all Demos," he presented the "oN-Line System" showcasing a scenario of an architect who designs a building using an application similar in structure to today's modern CAD[2] programs. His demonstrations contained so many groundbreaking innovations that many viewers had problems understanding what they saw, such as a display with multiple windows, mouse interactions

2 Computer Aided Design

for selecting objects and triggering commands, full-screen document editing, hypertext linking, e-mail, instant messaging, and even video conferencing.

However, neither the Stanford Research Institute nor Engelbart turned these novel ideas into commercial products. Instead, this was initiated by the photocopier company *Xerox*. In fear of Engelbart's vision of a paperless office, which would inevitably decline their paper-based business, the company decided to control this new technology so that it would not be overrun. Hence, Xerox formed the *Xerox Palo Alto Research Center* (PARC) in 1970 and gave top computer science researchers absolute freedom to explore whatever dreams they had about the future of computing. PARC soon became the hotspot of many novel, groundbreaking innovations - both in terms of software and hardware, e.g., the "Xerox Alto" - the first workstation with a graphical user interface using a typewriter-like keyboard, a three-button mouse, and a novel ethernet connection.

Moreover, with the first object-oriented programming language *Smalltalk* (see Figure 8.3), PARC researchers set the standard for a consistent user interface approach in 1974 by defining the *WIMP* interaction paradigm, describing windows, icons, menus, and pointers as the essential components of a graphical interface through which a user can interact with a computer system.

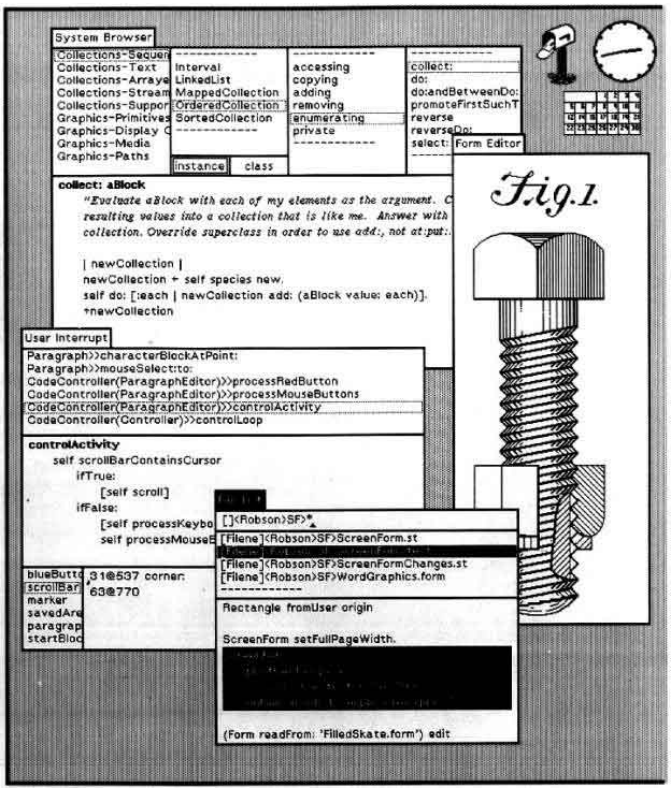


Figure 8.3: Smalltalk [154]

Although other companies like Apple and Microsoft took over during the mid-1980s to commercialize the ideas developed at PARC, it was Smalltalk and the Alto that, in 1974, defined how graphical user interfaces should look like, including the WIMP paradigm and many other of today's GUI elements, such as borders, title bars, overlapping windows, icons, popup menus, the desktop metaphor, scroll bars, radio buttons and dialog boxes, all of which remain standard in todays operating systems.

Natural User Interfaces

Besides the shift from CLI to GUI, PARC has also strongly contributed to the second major paradigm shift in human-computer interaction, which is from GUIs to *natural user interfaces* (NUIs). During the early 1990s, researchers at PARC worked on new ideas to break the boundary of desktop applications and envisioned several types of post-WIMP interfaces [68]. Their main intention was to develop new interaction concepts that better integrate with the *physical environment* and *real life* of the user. That is because standard GUIs rely on computer systems with keyboard and mouse and, therefore, inherently prescribe *single-user* application scenarios in which the user's freedom of interaction is limited to the desktop screen. To break down this limitation, *Mark Weiser* and other researchers at Xerox Parc envisioned a less demanding form of human-computer interaction. They called it *ubiquitous computing*, in which technology becomes "calm" by embedding it in the natural environment [283]. Instead of requiring the user to sit in front of a technical device and thus adapt to the technology in his natural behavior, in ubiquitous computing, it should instead be the technology itself that is integrated into the user's environment in such a self-evident way that it is no longer perceived as such [286].

Ubiquitous Computing

> *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."*
>
> — Mark Weiser [282]

However, to "disappear" into the environment does not necessarily mean being hidden from view. Instead, as suggested by cognitive scientist *Don Norman*, it is meant "to make technology conform to the needs of people" [195, p. 261]. For this purpose, it is necessary that the devices are conveniently at hand and do not require special effort for their use. Hence, being ready-at-hand and using devices without thought contributes to "disappearing" technology [139].

> *"Just as a good, well-balanced hammer 'disappears' in the hands of a carpenter and allows him or her to concentrate on the big picture, we hope that computers can participate in a similar magic disappearing act."*
>
> — Mark Weiser [281]

While the concept of ubiquitous computing transformed the fundamental understanding of human-computer interaction by envisioning a "proliferation of devices and systems that enable access to computation in a variety of ways" [139], it also was the driving factor for developing novel devices and technologies that built up on the user's given habits and capabilities making them "naturally" simple to use.

Today, these "Natural User Interfaces" (NUIs) "enable users to interact with computers in the way we interact with the world" [126] and represent the latest paradigm in human-computer interaction. Like ubiquitous computing, NUIs "aim to provide a seamless user experience where the technology is invisible" [126]. However, where ubiquitous computing delivered the overall vision of augmenting the natural environment with an interplay of various embedded technologies, NUIs can be understood as concrete instances of novel interactive devices designed to make use of naturally acquired human skills, such as touch, gestures, or speech [272].

For that reason, NUIs exhibit a high degree of heterogeneity compared to previous interaction paradigms. While CLIs are characterized by a keyboard input in a command line and GUIs are primarily oriented towards the WIMP scheme with a keyboard and mouse as input devices, the appearance of NUIs is manifold since they are not limited to particular input and output technologies. This circumstance also makes NUIs more challenging to define, especially by a simple one-sentence. This is also reflected in the fact that researchers still discuss what "natural" really means to specific audiences, for instance, children [51]. However, the following definition from the wiki of the *NUI Group*[3], an expert forum and online community for researchers in this domain, shall conclude this section:

"A NUI is an emerging paradigm shift in man-machine interaction of computer interfaces to refer to a user interface that [...] becomes invisible with successive learned interactions to its users. The word natural is used because most computer interfaces use artificial control devices whose operation has to be learned. A NUI relies on a user being able to carry out relatively natural motions, movements, or gestures that they quickly discover to control the computer application or manipulate the on-screen content." [250]

The following section will provide a short overview of the dominant input modalities to further illustrate the great variety and different form factors of natural user interfaces. By quickly elaborating on their individual strength and weaknesses and design considerations, it shall also serve as a starting point for the succeeding section about combining various NUIs to an *interactive space* that suits the special demands of agile Scrum teams.

3 The definition was originally accessible at [197]. Recently, the forum has been shut down because the host depreciated the community forum software.

8.2 DIFFERENT NUI TYPES

The following sections provide an overview of various NUI types.

8.2.1 *Touch and Multi-Touch*

Of all NUIs, touch interfaces are by far the most common and indispensable part of today's world, in which smartphones and tablet computers have become integral to daily life. While most people associate touch interfaces with these mobile devices, it is important to know that research has also long been concerned with other implementations, such as multitouch tabletops or large interactive walls, which are particularly interesting to this thesis.

Touch screens were first introduced in the 1970s

The history of touch and multitouch technologies goes back longer than one might think at first glance. Even before the development of the PC, early synthesizers and electronic musical instruments could be operated by touch-sensitive pads using capacitance sensors to manually control and influence the music produced [296]. Touch *screens* were first introduced to the public in the 1970s but could only detect and process a single touch point [180]. During the heyday of the PC in the 1980s, research was already beginning on *multitouch* displays, devices capable of recognizing multiple touches simultaneously and independently [45]. However, it was not until 2006 that the public became aware of this technology when *Jeff Han* presented the result of his research [111] at the famous TED conference (see Figure 8.4). In his talk entitled "The radical promise of the multitouch interface" [110], he demonstrated a "cheap, scalable multitouch and pressure-sensitive computer screen interface" and several application examples that caused a sensation among the audience.

Multitouch displays were researched in the 1980s but became publicly aware in 2006

Figure 8.4: Jeff Han at the TED conference[4]

4 Source: https://www.ted.com/speakers/jeff_han

Since the first iPhone in 2007, multitouch displays have become mainstream

Just one year later, with Apple's presentation of the iPhone in 2007, multitouch became suitable for mass use and, within a very short time, succeeded as the dominant technology for new devices such as smartphones and tablets. Next to these smaller devices, multitouch technology was also incorporated into larger displays. Also in 2007, Microsoft released the "Surface" - the first interactive tabletop capable of identifying various objects and their position on the screen besides sensing multiple fingers and hands. For Bill Buxton, this was "a key indication of this technology making the transition from research, development, and demo to mainstream commercial applications" [45].

However, to understand touch interfaces, it is crucial to know that their facets are extremely diverse:

> *"The term 'touch screen interface' can mean so many things that, in effect, it means very little, or nothing, in terms of the subtle nuances that define the [...] appropriateness of the design for the task, user, or context."*
>
> — Bill Buxton [45]

Characteristics of touch interfaces

At first, it makes a real difference in terms of the *directness* of touch, whether the interactive surface also is the display for the user interface or, in contrast, only functions as an input device (like touch pads in notebook computers) [45].

Besides the capability to display the interface, many more design considerations also depend on the given sensor technology of the touch surface. For instance, technical aspects define the *degrees of freedom* (DOF) and therefore limit the richness of possible interactions. Touch displays capable of detecting only one touchpoint have 2DOF and are similar to conventional GUIs, where a mouse pointer is moved on a 2D screen. In contrast, if a sensor allows tracking two fingers simultaneously, the user will already encounter 4DOF, allowing for a much richer interaction experience.

User distinction

Of course, tracking multiple fingers is not only useful for single-user applications but especially interesting for collaborative scenarios in which multiple users simultaneously work with an interactive surface, e.g., in the form of tabletops or wall displays. In these contexts, it would be valuable to know which touch point belongs to which person. This would allow to *distinguish between users*, for instance, to provide personalized user interfaces or access to user-specific data [297] but also to differentiate whether an interaction is executed by one person with multiple fingers or if multiple persons execute two (or more) concurrent interactions.

However, the capability of differentiating between multitouch and multi-user is still a subject of technological research. Either solutions rely on very specific hardware components or complex tracking algorithms, both having several constraints.

The MERL DiamondTouch system [74], for example, is capable of assigning touch points to different users on the hardware side but limits the freedom of movement of the users by physically wiring them to the device. Moreover, it suffers from "ambiguous responses when a single user exercises multiple contacts on the surface," which "limits the ability [...] to provide full support of common multitouch interactions" [46]. Other systems try to distinguish users by additional cameras for detecting their shoes [216], body [297], hand-contours [232] or incorporate additional wearable technology like smart-rings [24] or vibration-sensors [170] for user assignment. Without going into too much detail, the last thing to mention is the Fiberio system [118]. Its unique feature lies in a special technical structure that makes it possible to distinguish users and identify them by their fingerprints, which opens up new application possibilities [151].

Richness of touch data

In addition to the distinguishability of individual users, differences between touch interfaces can also be seen in the information they can obtain from a *single* contact point. In addition to recognizing that the surface was touched at a specific position (binary touch/no touch), some systems can also recognize the *degree of contact* and interpret it as variation in pressure or angle of attack and can calculate force vectors, all of which can be used to enrich the user experience with real-world gestures, e.g., flicking virtual objects on the screen [45].

Lack of precision

In addition to the extended possibilities offered by touch interfaces compared to classic GUI systems, there are also some disadvantages. Although object selection or manipulation with the finger is often superior to mouse interaction in terms of speed, it is clearly inferior in terms of *precision* [278]. Depending on the particular action, this lack of accuracy significantly affects the nature of the interaction. For instance, typing a particular letter on a smartphone keyboard already requires a high level of accuracy. However, selecting the gap between two letters to correct a misspelling is way more difficult. Contrasting to that, flick or slide actions, for example, used when swiping through images, are far less demanding and do not require interaction precision. In addition, precise interactions also demand for *greater visual attention* because of *missing tactile feedback*.

Missing tactile feedback

The lack of precision is also immediately apparent when experiencing the difference between using a graphical or physical keyboard. Whereas physical keys guide the user and make him or her feel their location, the absence of tactile feedback and haptic reference points requires the user to leave the eyes on a virtual keyboard. As Buxton pointed out, what is meant by missing "feedback" is indeed missing "feedforward," which he describes as "feedback for the task of finding the appropriate control, not activating it" [45].

This also relates to a significantly lower use of muscle memory, amplified by the fact that controls on touch interfaces are not persistent

in their location. Instead, several control elements typically appear at the same location at different times, increasing the user's cognitive load and limiting the potential for motor learning. Depending on the application scenario, the lack of haptics, together with the lack of precision, can be significant downsides.

Compensating the lack of haptics with digital pens and tangibles

For this reason, touch tablets are often equipped with digital stylus pens since they provide a much better user experience in the context of artistic drawing or even handwriting. Besides, some larger interactive surfaces, especially tabletops, make use of additional physical elements, so-called *tangibles* (see Figure 8.5), which not only compensate for the downsides of touch but also add new possibilities for both user input and system feedback.

Figure 8.5: Tangibles on the "Reactable" [132]

Next to technological differences, which already bring a great deal of complexity when designing touch interfaces, there are also very basic things to consider for appropriate usability. To name two, and beginning with the fact that arms and fingers are not transparent, touch interaction has to deal with the problem of *occlusion*. This is even more pronounced in collaborative work scenarios and increases with the number of people interacting on the surface simultaneously. In addition, collaboration on interactive tabletops suffers from an *orientation problem* since content (especially text) is not equally well recognizable from all sides.

As can be seen, touch interfaces have many different aspects to consider depending on the technology, context of use, application scenario, and much more, so what has been mentioned before is by no means complete. However, the purpose of this section is only to show that there is a lot more to the design of touch interfaces than to classical desktop GUIs because of the great variety of challenges and many considerations to take.

8.2.2 *Gestural, Speech, and Tangible Interfaces*

Out of the great variety of NUIs, touch-based interfaces are the most common. This section now looks at some other also well-known representatives.

Many applications require hands-free interaction with computer systems. For example, virtual reality (VR) environments try to avoid external controls to increase the immersive feel, thus making the user experience in the virtual world seem as real as possible. Clinical environments with crucial demands on hygiene could also benefit from no-touch interactions.

Gestural interfaces

In these scenarios, *gestural interfaces* might be a natural solution, where the computer system is operated by a set of gestural commands that are triggered by in-air movements of various body parts, commonly hands, arms, and fingers, but also head, legs, and feet.

From an interaction point of view, utilizing gestures that are executed freely in three-dimensional space opens up new possibilities. At the same time, however, it also represents the greatest challenge from a technical point of view. That is because the translation and mapping of gestures to commands is a challenging task due to the motion richness, including arbitrary poses, locations, and self-occlusion [263], which requires expert knowledge in computer vision and machine learning to develop algorithmic pipelines with high recognition accuracy [150]. In addition, as Liu and Thomas point out, designing gestures in a way that they are quickly and reliably recognized by imaging systems often comes at the expense of user effort. However, the effort greatly impacts a user's willingness to actually use a gesture-based system [162]. So, although usability studies have shown that some application scenarios have stronger user preference towards gesture interfaces as compared to touch interaction because of the overall more natural and intuitive feel [198], the trade-off between system reliability and user effort still is a major drawback.

Other usability challenges include *missing tactile feedback* [88] and the fact that simple pointing operations in gesture interfaces can be significantly slower compared to classic mouse interaction and may also lead to fatigue more often depending on the use case [47].

In summary, gesture interfaces in which interactions take place in free space offer a wide range of possibilities but present new challenges to developers, designers, and users equally. To better address these, Frederic Kaplan suggests that gestures and movements in space should not be understood as an emulation of a mouse pointer that can trigger a set of commands. Instead, gesture-based interaction is "a fundamentally different approach to the design of human-computer interfaces" in which the digital system must merge with the user's

physical environment to create a "halo" of interactivity [137]. This would lead to a reference shift that no longer positions the user in front of a digital system but instead adapts the interface itself so that the user can ubiquitously interact with interface components around his body.

Voice user interfaces

Another type of touch-less NUIs is *voice user interfaces* (VUIs), which enable communication between humans and machines through voice-based interactions, i.e., natural speech. Technically, VUIs are made possible through *speech recognition* and *natural language processing*, which allows interpreting human speech to understand verbal commands. In HCI research, VUIs have played a minor role for a long time compared to other interface types. However, with the release of voice assistants by major tech companies in recent years, e.g., *Siri* from Apple, *Alexa* from Amazon, and the *Google Assistant*, which rapidly gained in popularity, research activities in the domain of voice interfaces also seem to accelerate.

Beginning with the benefits of VUIs, the first one is that they allow not only a touch-less but also an *eyes-free interaction*. This suits many situations where the user would otherwise be considered occupied, for example, while driving a car or cooking a meal. In these scenarios, voice interactions can be more convenient than physical ones because they do not rely on visual attention or require hand availability. Likewise, to users with disabilities like visual impairments or limited hand dexterity who have difficulties accessing conventional display-based devices, VUIs can be the most efficient (or, in severe cases, the only viable) form of interaction [83]. However, as stated by Corbett and Weber, it is a general best practice of HCI and interaction design to advise against voice as a primary interaction modality since physical inputs are usually considered to be most efficient [63].

Apart from that, several fundamental and practical limitations in VUIs hinder the ubiquitous adoption of this technology [72]. For example, speech processing is often error prone [288] for several reasons. At first, making speech recognition systems robust to noisy environments is a significant challenge, so the performance of VUIs still heavily depends on the acoustic setting and cleanliness of speech data. Secondly, the error rate increases with a conversational and more casual style of speech [72]. That is because natural language is more than just words [13]. Free speech contains a lot of inexplicit utterances, such as ellipsis and deixis, which VUIs must be capable of dealing with [112]. In addition, language interpretation requires semantic knowledge, i.e., meaning, and pragmatic knowledge about the application context to achieve a common sense of what has been said [72].

All of this makes natural language interfaces not only difficult to engineer but also leads to frustration and confusion on the user side

because of unrealistic expectations regarding the technical capabilities of VUIs [183].

Complicating this, the *discoverability of voice commands* is a fundamental challenge to any VUI user. That is because users have inaccurate and often inconsistent mental models of what the system can understand. Hence, people must learn the speaking style recommended by the recognition software [203]. However, the ephemerality of voice input does not allow for sophisticated user guidance and assisted learning because of missing affordances, interaction metaphors, and the fact that commands are forgotten over time [63]. Moreover, the temporal and transient nature of audio makes listening to VUIs more demanding than visually scanning the screen of a GUI since information cannot be accessed directly or easily browsed [11] but has to be *remembered* instead.

Lastly, what further contributes to the fact that VUIs are challenging is that there are yet no established guidelines or taxonomies for assisting developers in building and designing more usable speech interfaces [184].

Tangible User Interfaces

While gestural and speech interaction has been subject to research for a longer time, another subcategory of NUIs has gained more attention in recent years. *Tangible User Interfaces* (TUIs) have been coined by *Hiroshi Ishii* and the Tangible Media Group at the MIT Media Laboratory in the mid-1990s [169]. According to Ishii, "TUIs represent a new way to embody Mark Weiser's vision of ubiquitous computing by weaving digital technology into the fabric of the physical environment, rendering the technology invisible" [123].

In GUIs, digital information is rendered on displays; thus, it is perceivable by vision only before being manipulated by mouse and keyboard, which act as general-purpose devices for interacting with all kinds of data. In contrast, the central idea of TUIs is to give digital information a concrete physical form that both serves as representation and control for its digital counterpart.

To illustrate this further, the *Marble Answering Machine* (MAM) will serve as an example. This conceptional device was presented in 1992 by *Durell Bishop* during his studies at the Royal College of Art and represented the very first concept of a TUI. Despite the age, it is still well-suited to demonstrate the benefits of the underlying idea, which is accessing digital information by manipulating physical objects.

Instead of displays, buttons, and dials, this answering machine incorporates *marbles* as central interaction objects. Whenever a caller leaves a message, a marble rolls out of the machine and lines up with other marbles representing previous calls, as shown in the upper right images in Figure 8.6.

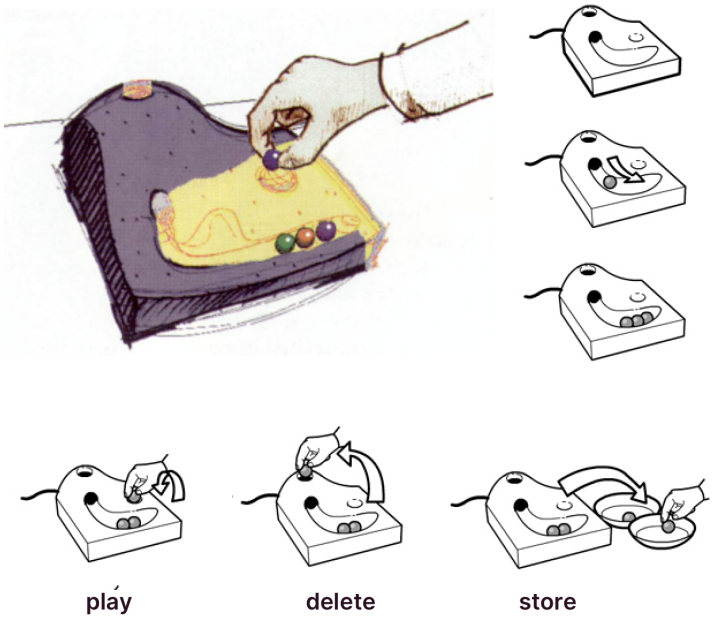


Figure 8.6: Marble Answering Machine[5]

The callee can then (1) *play* the message by taking the associated marble and laying it in an indentation, (2) *delete* the message by feeding the marble back into the machine, or (3) *store* the message by keeping the marble at a save place, such as a tray next to the machine.

But what is the benefit of the MAM when comparing this tangible design to a classical answering machine? To answer this question, imagine a person entering the room and briefly taking a look to see whether he or she might have missed some calls. Even if the classical machine uses a display showing the number of incoming calls, it would be difficult to read from a greater distance. In contrast, the MAM easily allows spotting incoming calls from further distance by just looking for marbles that came out of the machine. Suppose we further imagine that the color of a marble encodes a specific person. In that case, the MAM makes it possible to comfortably identify related calls and play messages in any order without additional effort. In contrast, the user of a classical machine would likewise need to jug through dials or operate the device through a bunch of button presses.

This example shows that TUIs can simplify interactions with a digital artifact (in this case, a telephone call) by making it accessible and directly manipulatable through an associated physical counterpart, a concept that Hiroshi Ishii has later coined "tangible bits" [124].

In addition, it also illustrates that TUIs are usually designed for *unique operations*. Deleting calls by feeding marbles back into the

5 http://dataphys.org/list/durrell-bishops-marble-answering-machine/

machine is a unique and very clear operation, so it is highly unlikely that the user will delete the message (feeding it back into the machine through a hole at the top) instead of listening to it (placing the marble into the indentation at the right side of the machine). As a result, the deletion of calls does not require extra confirmation, as is the case with classical answering machines, where the press of a button is a generic operation, so that incorrect actions are therefore more likely to occur, for instance, due to confusion. Lastly the MAM also demonstrates that TUIs have a unique potential for *user engagement* and *personal connections* since interactions take place in the physical environment and reality. In terms of the MAM, a person can, for instance, store all calls of his or her loved one by keeping the associated marbles in a place with personal reference. It would even be conceivable to form a necklace and "wear" the calls to create an exceptional personal connection. This kind of personal reference is hardly imaginable with a classical answering machine, where information and interaction remain in the digital space.

Overall, TUI's are a novel approach to human-computer interaction. Building on the foundation of ubiquitous computing (see Page 220) to make technology seamless and invisible, TUIs can hide the controls while simultaneously making the system's state evident on the surface as the physical form of the interface itself. Since research in this area has intensified significantly in recent years, it remains to be seen whether TUIs will no longer be counted as natural user interfaces in the future but will possibly trigger a paradigm shift of their own. In particular, Ishii's idea of Tangible Bits has also evolved into a vision of "Radical Atoms" [125]. These are TUIs that map the physical manifestation of digital data through shape-changing materials, allowing changes in the digital system to be reflected in changes to the physical form of the interface.

However, a significant disadvantage of TUIs is that each interface is custom-built and usually designed for one particular task only. This circumstance also makes it more difficult to establish standards, which in turn are necessary to promote the development and dissemination of TUIs. For this reason, voice-based, gestural, and, most importantly, touch interfaces will probably remain the predominant type of NUIs for the time being because they are much more versatile, and their types of interactions can be used for a variety of different purposes.

After explaining the major paradigm shifts in human-computer interaction towards natural user interfaces and presenting their main representatives, the following sections will now start drawing a connection to the identified Scrum issues and challenges of Chapter 6 by proposing an alternative solution based on touch-based NUIs.

Beginning with basic design considerations that help understand why the novel approach only incorporates touch-based NUIs, the

following sections will elaborate on important design considerations for individual interface types (mobile, tabletop, and vertical touch displays).

Afterward, a brief look at existing touch-based NUI approaches in the context of agile software development will be given, followed by an analysis of their weaknesses before proposing a novel system in Chapter 9.

8.3 BASIC DESIGN CONSIDERATIONS FOR A NUI SOLUTION

When developing a NUI-based approach for addressing the challenges of Scrum teams (see Chapter 6) and tackling the problems of GUI-based project management applications (see Chapter 7), the first thing to consider is which *type of NUI* to select. That is because, as explained in Section 8.2, NUIs come in great variety and different form factors, each with individual strengths and weaknesses. So, when deciding whether Scrum teams should be supported through gesture, voice, touch, or tangible interaction, some basic requirements and design considerations must be clarified first.

Simultaneous operation by multiple persons

First, the new system should be capable of being *operated by multiple persons simultaneously*. This requirement serves to support collaborative work situations better, as they arise in particular during the mandatory Scrum meetings, which have turned out to be problematic with regard to existing GUI-based agile ALM applications (see Section 7.3).

Natural interaction without additional components

What is more, *interaction should be as natural as possible* and should be performed *without additional technological components* only serving the need of the system to work properly. Examples of such components include, for instance, wearable gloves. In gestural interfaces, these gloves only serve as a workaround to increase the pointing precision and overcome the technical constraints of current visual tracking approaches. However, from the user's perspective, these gloves are an unnecessary obstacle that can impede the interaction flow [166]. For this reason, the system to build should be operated without any further components than what the user brings naturally[6].

Integrating with Scrum processes and workflows

Lastly, *technology must integrate appropriately* into the existing Scrum processes and workflows. This means particular attention should be paid to the fact that the individual meeting types of Scrum are carried out in different ways and may require various types of technological support and interaction design. When comparing, for example, the

6 In this context, "naturally" does not necessarily relate to body parts only but more to any object that is commonly part of the interaction environment. For example, a digitally enhanced pen to recognize and process written information can "naturally" sit in the environment of a sprint planning meeting. It may be used just like a normal pen and therefore becomes "invisible" or "natural" as a technological device.

sprint planning event with the daily Scrum, both meetings are different in terms of the content to discuss, duration, and overall setup. While the sprint planning event can take several hours and participants usually sit, the daily Scrum is supposed to be 15 minutes only and, therefore, is held as a standup meeting. These aspects must be considered when designing a usable system that adapts to the given work situations and processes instead of vice versa.

Based on these basic requirements, touch-based interfaces were selected for the system to be developed because they can fulfill all criteria, whereas other NUI types could easily be excluded because of significant disadvantages. Gestural interfaces have been considered inappropriate because the requirements above would demand a visual tracking technology capable of simultaneously processing gestures from multiple persons. However, this is not yet reliably possible. Likewise, speech interfaces have been excluded because they require just one person to speak at a given time. Moreover, speech processing is still limited to a command-based interaction, which would negatively affect Scrum workflows and meetings. Tangible User Interfaces, on the other hand, due to their free form of design and diversity, are certainly a suitable way to support the work of Scrum teams technologically in a new way. However, as stated on Page 230, each TUI is usually custom-built and designed for one particular task only. This circumstance makes TUIs less versatile for the complex work environment of agile Scrum teams, whereas touch interfaces can adapt to different tasks by switching the user interface on the touch screen.

8.4 DESIGN CONSIDERATIONS OF TOUCH-BASED INTERFACES

Given the decision to design a novel interactive system for agile Scrum teams that uses touch-based NUIs, the following sections elaborate more on important design considerations for three main touch-interface types.

8.4.1 *Mobile Interfaces*

According to Rogers et al., mobile touch-based interfaces refer to any computing devices designed for handheld interaction via touch input while being on the move [219]. While this definition generally covers a broader range of devices, e.g., e-book readers, the largest share is first of all represented by modern smartphones, which, after the introduction of Apple's iPhone in 2007, rapidly became the biggest trend in personal computing and quickly developed into an indispensable part of today's life [96].

That is because smartphones are way more than just mobile phones. Instead, they are internet-enabled computing devices with enough processing power and memory capacity to run all kinds of different applications, so-called apps. In terms of interaction, modern smartphones usually do not provide physical buttons but mainly rely on the finger as a natural pointing device on the touchscreen. In addition, current mobile devices include many different sensors, such as cameras, microphones, accelerometers, or near-field communication modules, all of which create many opportunities for novel applications and interaction techniques.

In general, the possibilities of app development for mobile devices are already very advanced. This includes, among other things, new development technologies and advanced APIs[7], as well as mature design standards for different operating systems that greatly simplify the programming of apps.

Design considerations

However, besides new possibilities, mobile devices are also naturally limited in various usability aspects. To further understand the possibilities and limitations of mobile interfaces, the most important design considerations are outlined below.

Limited display space

One of the biggest challenges in designing mobile interfaces arises from the *limited space* for displaying control elements due to the small screen size. This restriction could be circumvented by making complex action sequences accessible via multi-level navigation hierarchies. However, this also increases the cognitive load for the user [23]. Hence, Nielsen advises designing mobile interfaces list-like so that the action space can be increased at will via vertical scrolling. Principally, though, he recommends limiting the displayed functions and information to those that are most important for the current use case [194]. Another piece of advice for dealing with the problem of limited display space is given by Shneiderman and Plaisant, who recommend using established design patterns and visualization techniques for grouping information and creating a more streamlined mobile experience. However, they generally recommend optimizing mobile interfaces for short and simple tasks while simultaneously moving complex and longer tasks to the desktop whenever possible [252].

Occlusion and less precise touch input

This recommendation also makes sense because the already small action space of the mobile display is further reduced by *occlusion* of the fingers. After all, whenever a finger touches the surface, it covers a visible part of the screen. Due to this relatively large contact area, finger interaction is also *less precise* in comparison to the pixel-precise pointing with the mouse when using desktop computers. As a consequence, although the finger as a pointing device is more intuitive and provides a "stronger feeling of having control over the interactions"

7 Application Programming Interfaces

[14], mobile interfaces can be more difficult to use, which is especially true for users with poor manual dexterity or bigger fingers [219]. For this reason, Nielsen recommends that mobile applications should not demand pixel-level precision and that interactive components should be sufficiently large [194].

Lack of secondary interactions and hover states

Besides occlusion and lack of precision, touch interaction also lacks other features that are an integral part of classical WIMP interfaces. For example, the mouse usually has additional controls, such as buttons and clickable scroll wheels, which trigger predefined secondary control options like opening context menus. However, the finger is a pointing device without additional control "features." Moreover, it naturally inhibits the rendering of a cursor on the screen and consequently does not allow the detection of hover states, further limiting the capabilities of interface design [190].

Gestures

On the other hand, interaction designers of touch interfaces can take advantage of *gestures*, which "add a welcome feeling of activity to the otherwise joyless ones of pointing and clicking" [196]. However, with respect to gestures, there are some important aspects to be aware of. One is that gestures have no visible signifiers. On the positive side, this benefits the limited size of mobile touch screens and prevents cluttering the UI with further controls. On the other hand, the absence of visual clues means that the user has to deduce the possible space of actions on his own. It also means that gestures pose more cognitive load on the user because they have to be memorized and cannot be accessed through visual cues in the UI. Consequently, gestures should be as self-revealing, intuitive, and easy to learn as possible [14]. While this is feasible for simple tasks like zooming, rotating, and swiping, it is, however, very challenging for complex tasks, which is a further reason to leave those tasks for desktop computers.

Software keyboards

Besides pointing, mobile touch interfaces are also highly different in terms of *typing*. In contrast to desktop computers and laptops, smartphones do not have physical but solely rely on software keyboards, which are less satisfying because of several downsides. At first, text entry on a software keyboard cannot match the speed and efficiency of the physical pendants [219]. In addition, the lack of tactile feedback also leads to higher error rates [80], which in the case of smartphones are even worse due to small key sizes combined with occlusion, as mentioned before. Furthermore, software keyboards claim a reasonable portion of the screen during text entry, further limiting the application's viewport. As a result, the user cannot see more than a limited portion of the written text. For this reason, text entry on smartphones should be reduced whenever possible, and applications demanding higher amounts of text input should better be brought to the desktop [252]. Nonetheless, as Nielsen stated, text entry remains an integral part of many mobile applications and, therefore, should

be enhanced through algorithmic approaches, such as auto-complete, type-ahead search, or automatic spelling correction for a better user experience [194].

Context of use

In addition to the points mentioned so far, mobile devices also differ in how they are used in different *contexts*. In contrast to desktop computers and laptops, the context of smartphones changes more frequently because "mobile" also implies that these devices are ready at hand and, hence, used in many more situations under diverse ambient conditions. That is why mobile users are also more easily distracted due to various external influences, e.g., noise or interruptions from other persons. As a consequence, they usually have less time to perform a task, which could explain why mobile browsing is observed to be shorter and more goal-directed, whereas it is less-directed ("surfing") in the case of desktop computers [266].

Apart from that, the context also determines how tasks are executed on a mobile device. Task performance may, for instance, depend on the location, so users need different functionality for a given task depending on whether they are in their office at work or traveling by train. For this reason, Benyon promotes context-aware mobile interfaces, which can adapt to the current situation and task at hand by providing context-sensitive functionalities depending on the location, the computational environment, and the history of interactions with the environment [23].

8.4.2 *Tabletops*

Interactive tabletops have gained more attention in the research community since 1993 when *Pierre Wellner* introduced the *DigitalDesk*. This computationally enhanced desk was operated via touch gestures and could augment physical paper with digital information through a top-projected interface [284]. Since then, and especially throughout the last ten to fifteen years, interactive tabletops have become a substantial topic in HCI research, especially in relation to co-located activities and collaborative work scenarios.

Out of all interaction devices, the *table* probably feels most natural to humans since we incorporate tables as a place to meet in our daily lives [108]. From eating dinner with friends to collaborative business activities with colleagues or customers, tables are an indispensable part of our social interactions. As part of this, we share verbal and non-verbal communication cues while interacting with objects on the surface, like, for instance, paper.

The same holds for interactive tabletops, which, similar to their physical pendants, also represent a place to meet, while their large

multitouch surface simultaneously extends the capabilities of ordinary tables by providing a shared digital workspace, which can be operated by multiple persons simultaneously. Naturally, this makes interactive tabletops particularly suited for all kinds of collaborative activities, which benefit from higher awareness about each other's interactions. However, while tabletops offer many opportunities and affordances, they pose additional challenges to interface designers [241].

Design considerations

Regarding basic touch input, it could be assumed that the design considerations for mobile interfaces also apply to tabletops since the problem of occlusion and less precise input as a result of the finger as an input device should remain the same. However, as pointed out by Watson et al., the larger interaction space of tabletops allows designers to better compensate for these problems, for example, by adding more complex multi-finger gestures to the interaction set, which, in contrast to mobile interfaces, are simpler to perform on tabletops because of the bigger interaction space [279]. What is more, they report that "both speed and accuracy improved when using the [tabletop] multitouch display over a mouse" and that "participants were happier, more engaged" and generally felt more competent as well as more immersed.

Software keyboards

Regarding text input, however, tabletops are equally limited as mobile interfaces and not suited for text-heavy tasks. The bigger display space does not contribute to the overall downside of software keyboards, which is a lack of tactile feedback that provides a significantly worse user experience than physical keyboards. As a result, Ryall et al. state that "providing virtual keyboards on the tabletop has proved a feasible, but tedious, solution" [228] and thus indicate that software keyboards are only used for lack of alternatives.

Shared input

Similarly, designers should generally avoid using traditional WIMP components in tabletop applications or, even worse, not just run existing WIMP applications on the tabletop. While this is technically possible, it should be clear that UI components initially implemented for the desktop and designed as single pointer interfaces cannot exploit the capabilities of *shared input* for which tabletops excel. Switching between different collaboration styles, e.g., working in parallel, sequentially, or under assumed roles, is what people naturally expect when collaboratively interacting with artifacts on the tabletop surface. For these natural workflows and group dynamics, tabletop interfaces must support *concurrent multi-user interaction* because this is what allows teams to focus on the given tasks and to choose among various interaction styles instead of being forced to coordinate and take turns [108]. Technically, shared input may pose particular challenges for designers depending on the work scenario. While tabletops provide multitouch technology for shared input, this does not imply that touch points can be associated with individual users. Identifying and tracing personal

Concurrent multi-user interaction

interactions can (depending on the actual tabletop technology) be very challenging [252]. That is because authorizing touch points to individual users usually demands special hardware [74] or complex workarounds [1]. However, these are only necessary if the application scenario requires unique assignments of touch points to users, which is mostly not the case.

Fluid transitions

In addition to shared input, Scott et al. recommend designing collaborative tabletop interfaces for *fluid transitions* between various input modes [241], e.g., textual input for text entry or positional input for drawing on the surface. Switching between various activities with little or no overhead would allow users to focus on communication instead of operating the interface. For doing so, they propose universal input devices, e.g., a stylus, which can be used for multiple activities, e.g., text entry and drawing, and therefore avoids switching between different input techniques. However, in terms of tabletops, using a stylus is rather uncommon due to technical limitations. That is because while writing with a stylus, the ball of the hand usually touches the surface for ergonomic reasons. However, most systems cannot differentiate what is actually causing a touch point, thus leading to many false inputs, which may heavily affect task execution.

Orientation and readability of text

Another design consideration special to interactive tabletops is the interface's orientation. When interacting with a tabletop during group work, people naturally distribute across all sides and hence do not share a common perspective. This *problem of orientation*, i.e., viewing the interface from various angles, fundamentally differs from classical desktop GUIs and presents designers with major challenges [108]. In particular, this problem occurs with any presentation of text because, for proper text reading, it should be aligned with the direction of the reader's gaze, which, however, is impossible to achieve for multiple people spread around the tabletop sides.

The simplest solution to manage orientation while using digital tabletops is to let users manually rotate items on the surface [231]. However, designers must decide on an interaction technique even for this simple task of rotating objects. It may be as simple as adding rotation handles to any object on the surface or more complex like the *Rotate 'n Translate (RNT)* technique [149], which uses multitouch gestures and physical calculations for simulating surface friction to mimic the experience when manipulating physical documents on a usual table. In any case, it is important to know that manually rotating digital objects on tabletop surfaces can be more difficult to use and time-consuming when compared to traditional media [148].

For this reason, several *automatic orientation techniques* have been proposed in order to rotate an item towards the reader and therefore minimize manual rotations, e.g., *TNT* by Liu et al. [160] or *Vector Fields* by Dragicevic and Shi [78]. Technically, these techniques rely

on either the position of the document itself, e.g., by assuming that documents positioned closer to one side of the tabletop should be rotated to the side accordingly, or the position of the user, which may be calculated from the touch point vectors of fingers and wrists [231]. Both approaches, however, cannot solve the problem of the resulting orientation not being appropriate for shared objects and users distributed along the tabletop sides. However, according to Scott et al., automatic orientation can be a good solution for private work objects or personal, user-associated regions on a tabletop [241]. For cooperative use cases, though, techniques of manual object rotation remain the best choice. That is also because they enable users to facilitate orientation as a resource for group interaction since "collaborators often use [a] temporary and partial rotation of objects for communicative purposes, such as directing the group's focus, sharing information, and assisting others" [241].

Physical dimensions

Regarding group interaction, it is also important to consider the *physical dimensions* of a tabletop because it relates to the size of the group, its dynamics, and social behavior. While the number of people comfortably fitting around a tabletop is naturally limited, it is also restricted due to social norms since people need a sufficient amount of personal space to maintain a feeling of privacy [218]. In addition, people have a strong tendency to work at arm's length and try to avoid reaching into someone else proximity to respect their personal space [241]. According to Haller et al., it is therefore crucial for active group participation that designers have considered closer interaction proximity, i.e., objects should be in direct reach, and users should not be forced to reach across the whole tabletop in order to perform an action [108].

Individual vs. group activities

Furthermore, it is important to know that people commonly switch between *individual and group activities* [108]. Hence, tabletop interface designers should consider this behavior and enable both personal and group work, for instance, by dividing the tabletop surface into private and shared territories [240] or providing further displays in addition to the group interface. The combination of tabletop and wall displays is particularly suitable for this purpose because it allows various collaboration styles and enables fluid transitions in group work [128].

Overarching workflows

Fluid transitions also play an essential role in integrating collaborative tabletop tasks into *overarching workflows*. While the tabletop might be the interface of choice for collaborative work, a classical WIMP interface might suit better for related subtasks or even overarching activities [85]. Therefore, designers should enable fluid transitions between tabletop collaboration and personal work. This includes persistent data provisioning for further processing the collaborative work results at desktop computers [108], but also incorporating external

work into the collaboration by allowing attendees to use externally generated artifacts on the tabletop [241].

8.4.3 *Vertical Displays*

Similar to interactive tabletops and the rapid development of multitouch technologies, multitouch-enabled *vertical displays* also gained in popularity during the last ten to fifteen years within the research community. Given the fact that we are used to interacting with physical boards in our everyday lives, it is hardly surprising that interactive displays quickly emerged as successors of existing traditional whiteboards.

Visibility and informational awareness

Compared to horizontal tabletops, vertical interactive displays provide similar input capabilities, e.g., they also offer a large interaction space and are equipped with a touchscreen, enabling multiple users to interact with the display simultaneously. However, the vertical arrangement also results in fundamental differences, for instance, better visibility of displayed data and a shared view to distant users, which is an important aspect, especially for collaborative work scenarios. Better visibility also reflects in the fact that vertical displays have more capabilities to raise *informational awareness*, which is why they are often used for monitoring and dynamically displaying recent information [252].

Design considerations

Moreover, vertical displays lead to a different perception and usage of the space of action due to social conditioning [108] because people have different use cases for physical tabletops and whiteboards, which also project to their digital counterparts. As a result, vertical displays call for individual design considerations, the most important of which are outlined in the following.

Consistent orientation

While there is an orientation problem for interactive tabletops, there is generally none for user interfaces of vertical displays. That is because, due to their alignment, all users share the same perspective, making vertical displays more visible to a larger audience and readable from a greater distance. According to Haller et al., the alignment makes the interface design also closer to the traditional WIMP approach [108]. However, this must be questioned because, regarding the means of input and display size, special design considerations must be taken for interactive vertical displays.

Gorilla arm effect

Probably the most profound is that vertical arm movements are way more demanding than arm movements on a horizontal tabletop surface. They quickly cause fatigue and "a feeling of heaviness in the arm," which has been coined the "gorilla arm effect" [117].

While this name sounds amusing, the impact of the "gorilla arm" is substantial and should not be underestimated, as shown by a study of Pedersen and Hornbæk, in which 13 of 16 participants preferred working on a horizontal tabletop, because "[the] vertical surface was found physically more demanding to use" [207]. Further complicating this issue is the study of Kajastila and Lokki showing that arm fatigue even occurs when the vertical display is only operated for a short period of time [134].

Software keyboards

Consequently, Shoemaker et al. suggest using vertical displays together with alternative input concepts, preferably those that can manage interaction without vertical arm movements like mobile devices or horizontal mid-air gestures [253]. Moreover, since software keyboards are unsuited to be operated vertically, they also propose that textual input should be reduced to an absolute minimum, and software keyboards should be replaced by alternative input options, such as pen-based input or mid-air input techniques.

Reachability

In addition, touch interaction can also be challenging because of the sheer display size. This is especially true for very large vertical touch displays, also called "wall displays." These are much bigger than usual desktop displays, so the *reachability* of user interface elements can, therefore, become an issue [108]. As a result, the user experience can suffer from far-reaching arm movements when particular action sequences require the user to interact with widespread user interface elements.

Audience sizes and visibility aspects

However, a particular benefit of these large wall displays is that "they can accommodate groups that are likely to change in size, and where information that needs to be shown and discussed is to an audience of people" [218]. Compared to tabletops, where the number of people who can comfortably view the surface is limited to only a few, vertical displays have a clear advantage. However, as the number of group members increases, the distance between participants and the display also increases, which in turn results in visibility problems, especially with regard to the readability of texts. Therefore, it is important to consider the size of user interface elements with regard to the expected distance of the audience, which is especially true for *informational displays*, whose primary purpose is to provide a shared view for distant users.

Less interactive collaboration

Another important aspect to consider is that vertical displays seem to promote a weaker kind of collaboration, which is less interactive compared to tabletops. In their study about the effects of the physical affordances of interactive displays, Rogers and Lindley observed different behaviors in terms of group work depending on whether a group was using vertical interactive displays or horizontal tabletops.

While the tabletop groups "switched more between roles, explored more ideas and had a greater awareness of what each other was doing," groups with the vertical display condition found it more difficult and uncomfortable to work closely together and described their experience as "awkward" [218]. Moreover, vertical displays seem more likely to disturb collaboration because interacting persons tend to switch into a relatively persistent presenter role and turn their backs on the rest of the group. According to Rogers and Lindley, this behavior moves others out of the focus of attention, which makes it difficult to maintain group awareness.

As a result, it can be concluded that vertical displays are, in contrast to interactive tabletops, rather unsuitable for group activities that require close collaboration. Instead, Rogers and Lindley suggest that they are better at "providing a shared surface for communal and audience-based viewing and annotating of information that is to be talked about and referred to" [218]. This is also confirmed by Haller et al., stating that vertical interactive displays are "well suited for presentation tasks" and for "displaying information that are relevant for everyone [in the present group]" rather than for tasks requiring concurrent multi-user interaction [108].

8.5 RELATED WORK: TOUCH-BASED NUIS IN AGILE SETTINGS

While touch-based NUIs have been the subject of numerous research in recent years, their intended use within agile working scenarios is relatively unexplored. This section provides an overview of the work to date.

8.5.1 *AgilePlanner*

The *AgilePlanner* system shown in Figure 8.7 was developed by Liu et al. as "an environment for collaborative iteration planning" [161].

It was designed for two activities. The first one is referred to as *user story management*, which the authors describe as a collaborative process in which the attendees would usually use pen and paper (especially sticky notes) for gathering and defining product requirements from the user's perspective. Building upon this principle, the authors substitute this "pen and paper" metaphor with digital pens and tablet computers. That way, each attendee uses a digital pen to write stories on a tablet computer through an input form provided by a custom "MASE" application.

This application also serves as the system's backbone, so user stories created on these tablets automatically show up on a digital tabletop

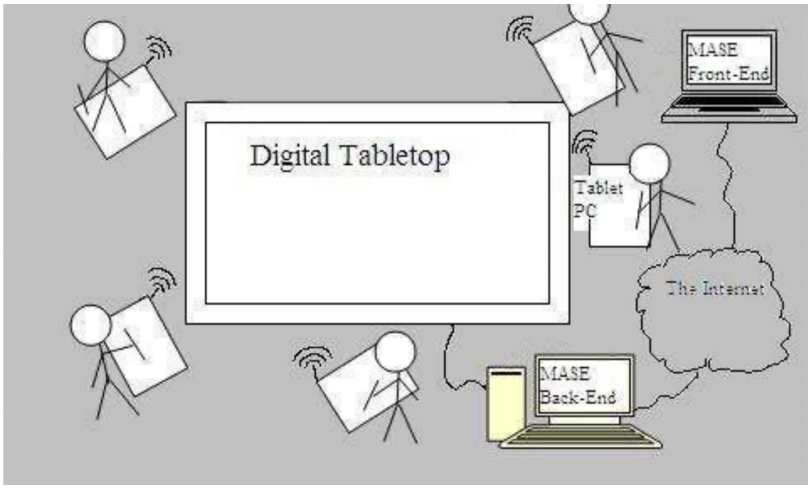


Figure 8.7: The "AgilePlanner" system [161]

representing a *shared workspace* for the second activity, referred to as *iteration management*. For this purpose, the user stories previously created on the tablets can be moved with the fingers on the tabletop and arranged collectively. A particular area on the tabletop serves as a container for the sprint backlog so that all stories placed in this area are considered to be included in the next iteration.

While the authors claim to have created a "fully digital collaborative planning environment" [161], they likewise remark that the system only represents a proof of concept since it has not been evaluated in any way nor investigated in terms of its usefulness.

8.5.2 *Agile Planner for Digital Tabletops (APDT)*

The *Agile Planner for Digital Tabletops* (APDT) by Weber et al. is described as "an advanced prototype that applies tabletop technology to support collocated and distributed agile planning meetings" [294]. It allows for multimodal interaction, i.e., besides touch, APDT also incorporates mouse and keyboard interaction, digital pen input, and voice commands to interact with the tabletop [280].

Regarding the usage scenario, APDT solely focuses on emulating agile planning meetings and thus tries to establish an orientation-independent workspace for discussing digital user story cards. In this process, the user is able to perform basic operations, such as creating, selecting, moving, rotating, and deleting cards on the tabletop.

These operations can be triggered via touch interaction or voice commands (in the case of the create and delete operations). In addition, the system allows cards to be labeled using either fingertips or

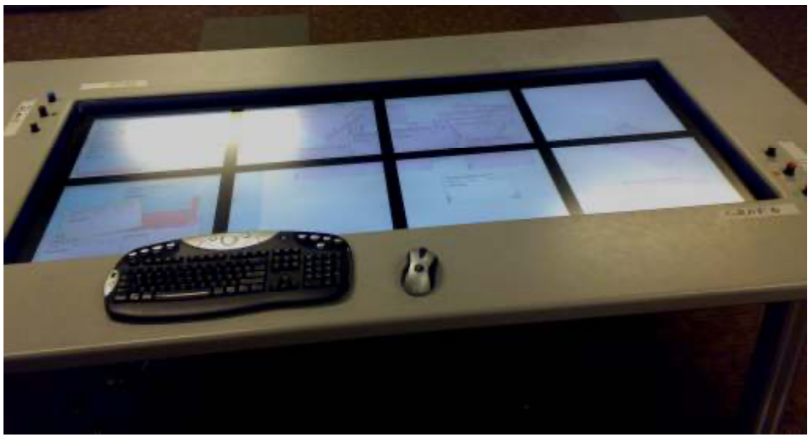


Figure 8.8: Agile Planner for Digital Tabletops (APDT) [280]

electronic pens. In both cases, the strokes are converted to text by a handwriting recognition engine [94].

During their evaluations of APDT, the authors focused on two questions. The first was about whether the system could facilitate agile planning meetings, which was investigated through observations and survey responses from usability tests with nine individuals who had to complete fifteen predefined tasks [294]. Based on this, the second question was about the system's usefulness in agile planning contexts [94] and whether it would maintain or change the behavior known from traditional pen and paper-based agile planning meetings [294].

The results show that the tabletop was generally perceived positively for the scenario of agile planning. However, according to their observations, the authors remark that most of the time of the collaborative meeting is actually spent on discussions rather than operating the tabletop surface and interacting with digital artifacts. While tool interaction is minimal, it is nonetheless crucial to provide a seamless user experience with great usability so that face-to-face communication is guaranteed not to be disturbed.

However, according to the results, APDT could not meet these requirements due to usability shortcomings and technical limitations. For instance, the participants complained that the implemented finger gestures were not easy to remember, thus increasing the cognitive load instead of providing a better workflow. Simultaneously, the experience of mutually manipulating the digital artifacts was low since the system only allowed processing two touchpoints at a given time [294].

Moreover, the other input modalities and, thus, the multimodal approach itself also revealed some major downsides. Voice control was not considered useful in such a collaborative work scenario because it necessitates quiet environments and thus implies a very orderly behavior of the participants, which poses an obstacle to free face-to-

face communication. Similarly, the handwritten editing of the cards was also declined because it felt unnatural to write with the fingertip. Writing with the electric stylus was also considered less helpful because resting the wrist on the surface caused false inputs and had to be prevented carefully.

8.5.3 *Ambient Surfaces: Interactive Displays in the Informative Workspace of Co-Located Scrum Teams*

In 2016, Schwarzer et al. investigated how so-called *ambient surfaces* might influence the work of co-located Scrum teams [238].

Based on the premise that permanently displaying relevant information about the development progress would be beneficial, the authors conducted a long-term study with a software development company using two multitouch displays that were installed in a freely accessible location next to a shared printer within the workspace of several Scrum teams. On these displays, the teams had permanent access to charts visualizing sprint progress, the teams' wiki system containing declarations and announcements, status reports from the test system, and an overview summarizing build failures and errors, all of which they could interact with using their fingers.

Although all of this information was already available to the individuals through programs on their desktop PCs, and they would have been able to display various types of data within a dashboard, for example, it became apparent that these ambient surfaces could additionally enrich the information workspace and thereby increase awareness within the Scrum team.

Over the course of two years, the authors observed how people used their system and investigated how it might influence a team's understanding of the development process. As it turned out, usage occurred during the daily Scrum meetings and mainly in informal situations, e.g., during lunch breaks or while waiting for team meetings to begin. In these situations, it could be observed that a single person interacting with the displays automatically attracted others to join, which led to spontaneous discussions and, as a result, enriched communication about the development progress among the team members. This result was also evident in the study survey, where over 90% of the participants said they frequently discussed issues with other people in front of the system. However, in terms of interaction, the overwhelming use was passive, i.e., people looked at the displays but rarely actively operated them with their fingers.

Overall, the ambient surfaces of Schwarzer et al. seem to be a simple yet promising approach to enriching the information space of

co-located Scrum teams. As perceived by three-quarters of the participants, the system increased visibility and provided easy access to information about the team's development progress, thus contributing to becoming more aware of what others were doing.

8.5.4 *The dBoard: A Digital Scrum Board for Distributed Software Development*

The *dBoard* by Esbensen et al. is a combined digital task board and videoconferencing tool for distributed agile software development teams [82]. It consists of a 65" multitouch display, a camera and microphone for capturing video and audio, and a sensor to detect the proximity of users. Two dBoards installed at distinct locations can establish a permanent connection in the form of a "virtual window" between distributed teams by superimposing a shared task board on top of videoconferencing (see Figure 8.9).

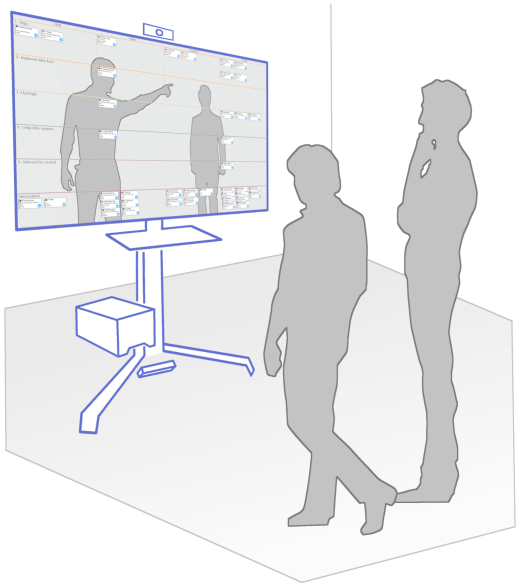


Figure 8.9: The "dBoard" system [82]

According to the authors, the background video stream shall provide the same feeling as "looking through a window into another office" [82]. Simultaneously, both sides are able to operate the superimposed task board, which appears to levitate in mid-air between the rooms (see Figure 8.10).

By reassembling traditional physical task boards, tasks in the dBoard are represented as small digital sticky notes. By touch, these can be moved across the board, assigned to developers, filtered within the board, and opened to access more detailed information, such as the task's description, time estimate, the work state, and change history.

Figure 8.10: Tasks shown on the dBoard [82]

Thereby, system states are kept synchronized across the two connected sites. As a result, tasks filtered on one board will also be filtered on the other side. Moreover, UI states are also synchronized so that touch points on either side are visualized as small red pointers on both dBoards, just like a task is highlighted in red when touched on either side.

With this feature set, the authors describe the dBoard as an "active meeting support tool" during daily Scrum meetings [82]. Furthermore, by visualizing the current work status as a task board, the dBoard is supposed to act as a "passive information radiator." Lastly, the system is intended to establish an "immersive interactive media space." What the authors mean by that is that the connection between two dBoards is permanent, so both sides are always on. Video and audio are automatically captured when walking up to a board by proximity-based interaction, which should enhance awareness about the presence of remote team colleagues.

While the evaluation results show neutral responses regarding possible performance improvements, the overall usefulness of the system was rated positively. Though people were unsure if the dBoard could speed up development, they rated touch interaction on the board as easy to use and very helpful. They also perceived the combination of videoconferencing and Scrum board as very useful.

8.5.5 *A Cooperative Multitouch Scrum Task Board for Synchronous Face-to-Face Collaboration*

The *Multitouch Scrum Task Board* by Jessica Rubart is a proof of concept and has been designed for organizational and planning activities within daily Scrum meetings of co-located teams [225]. In contrast to others, Rubart's task board does not target vertical displays but large interactive horizontal tabletops.

In terms of functionality, the system is, however, rather limited. By touch, tasks can be moved between different columns representing their status of work, such as "To Do," "In Progress," and "Done." Through common gestures, such as pinch-and-zoom, tasks can further be resized and rotated by multiple users simultaneously. New tasks can be created and assigned to a work status by double-tapping the respective column, and the new tasks will show up afterward. For further actions, a simple tap on a task opens a menu offering possibilities for deleting a task, splitting it, or specifying details via on-screen keyboards.

Besides task actions, the board also offers reporting views in the form of documents containing burndown charts and the number of task splits, which, according to Rubart, shall help evaluate the life cycle of tasks during sprint retrospectives. These documents can be moved, resized, rotated, and visually arranged by multitouch gestures, as well as annotated by touch and finger strokes, to support group discussions during sprint analysis.

While Rubart's system only represents a proof of concept for applying horizontal interactive tabletops to Scrum task boards and using common multitouch gestures for cooperative group settings, the usability evaluation brought two key insights. As expected, the common multitouch gestures for moving, resizing, and rotating tasks were well received. However, editing tasks via on-screen keyboards was considered difficult and slow. Although all test persons have been used to on-screen keyboards from their personal smartphones, the bigger pendants caused usability problems on horizontal tabletops and were considered negative. Furthermore, the evaluation revealed a *need for coordination* when performing certain tasks. Creating a task, switching between task and report views, and annotating documents all required a certain amount of agreement in terms of responsibility and coordination in terms of actions to avoid getting in the way of other group members.

8.5.6 *Nori Scrum Meeting Table*

Voss and Schneider presented *Nori* as an interactive, multitouch-enabled meeting table supporting the whole Scrum software development process [274]. However, this statement must be viewed critically since the range of functions and usability are severely limited.

Regarding its features, the tabletop is restricted to basic backlog and team management functionality. In the backlog view, cards representing user stories can be created via touch and specified in more detail using an on-screen keyboard. Interestingly, backlog items cannot be moved or freely arranged in the backlog view but are bound to a predefined grid layout, which limits the number of collaborative interactions. For prioritizing backlog items, the backlog can be switched into another view (see Figure 8.11), in which cards can be freely moved into one of three different areas representing different levels of priority (low, medium, and high) so that cards moved into these areas are prioritized accordingly.

Like the backlog, the team management feature offers two distinct views. The first shows team members as avatars, which can be freely arranged on the surface. A team is created by touching a special team symbol with one finger while drawing lines to each avatar with a second finger to create a connection between team members. In the second view, similar to the prioritization of backlog items, team members can be associated with various roles by moving avatars into distinct areas on the tabletop surface.

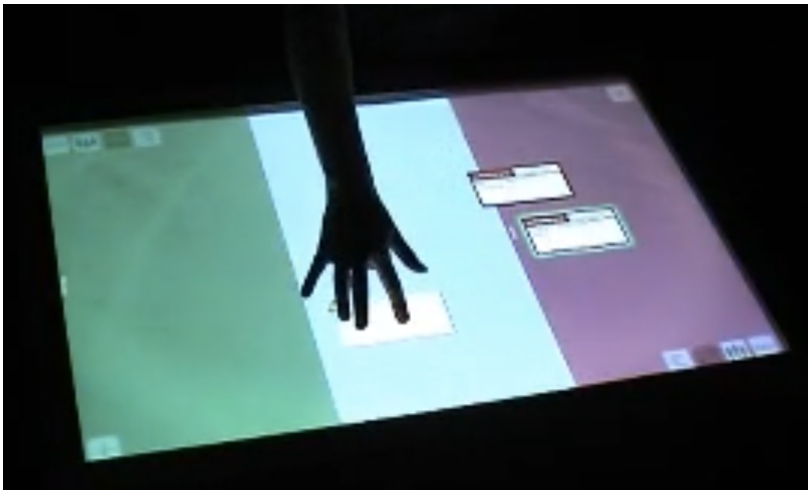


Figure 8.11: The "Nori" system [274]

Although the authors claim that the interactive meeting table would increase the efficiency of Scrum teams in a software development process [274], its usefulness must be doubted since the feature set described is by far not enough to support Scrum teams and targeting

the challenges identified in Chapter 6. In addition, Nori's demo video shows two main drawbacks in terms of its usability. At first, elements on the surface cannot be rotated or arranged freely to compensate for different viewing angles of persons standing around the table. Instead, the system allows the orientation of all elements to be flipped so that only two opposing sides of the table are allowed to read texts properly, whereas the other two sides are neglected. Secondly, the system also shows technical weaknesses, such as long latency during touch interactions as well as faulty touch detections and recognition errors, which limit the system and its interaction experience to such an extent that it would hardly be accepted in practical use.

8.6 WEAKNESSES OF CURRENT APPROACHES

Comparing the research work of touch-based NUIs in agile environments with the many problems identified in leading agile ALM applications (see Chapter 7) shows that GUIs and NUIs share the same weaknesses.

Technical weaknesses

On the GUI side, *technical weaknesses* are less pronounced because of existing and mature technological standards. However, as seen in Section 7.2.2, outdated web technologies are still common, especially in applications with a longer history on the market, and may result in slow and non-responsive interfaces, strongly affecting a team's work efficiency.

In contrast, technical weaknesses seem to be a more substantial problem in current touch-based NUI approaches. The APDT system, for example, is limited to two simultaneous touchpoints. Moreover, handwriting with a digital pen causes wrong inputs while the wrist is laid onto the surface, triggering false touch events (see Section 8.5.2). Both problems result from insufficient technology, just like the unreliable touch-processing of the Nori system, which has long latency and unstable gesture recognition (see Section 8.5.6).

Overall, these technical drawbacks are not uncommon and can be found in many of the bigger touch surfaces. That is because bigger surfaces are usually not built with the same precise and reliable capacitive touchscreen technology known from today's smartphones and tablets. Instead, for reasons of costs, bigger surfaces typically rely on optical touch tracking systems, which are significantly cheaper but also less error-prone[8]. As a result, current NUI systems using such optical tracking mechanisms are not technically mature yet and, hence, often only represent rudimentary prototypes.

8 More discussion on this topic can be found in Chapter 9.2.2, where a custom-built tracking system will be introduced that overcomes many of the existing technical limitations.

Poor interface designs

Another weakness shared by both the GUI and NUI interaction paradigms lies in *poor interface designs*, which can create severe usability problems, highly limiting a system's overall usefulness. In the case of the investigated agile ALM applications, it was, for example, shown that inconsistent or cluttered UIs cause unnecessary cognitive load on the user (see Section 7.2.2.2). Likewise, on-screen keyboards cause usability problems on horizontal tabletops. This is especially problematic when systems such as the Multitouch Scrum Task Board (see Section 8.5.5) are designed around the central action of editing interface components via the on-screen keyboard, thus making it a very common task. In addition, this system also showed that cooperative working practices with multiple persons interacting simultaneously place special demands on the interface design. Coordination in terms of actions between people must be considered to avoid people getting in the way of others when interacting with the touch surface. This also includes, for example, that the interface is not limited to individual viewing angles, as in the case of Nori (see Section 8.5.6).

Limited functionality

Lastly, the current systems are also severely limited in terms of *proposed functionality*. In the case of GUIs and the investigated agile ALM tools, Chapter 7 revealed drawbacks and a lack of features, especially in collaborative activities like the mandatory Scrum meetings. Partly, and as explained in Chapter 7.3, these limitations stem from the fact that desktop GUIs represent single-user interfaces. They, hence, are naturally limited for true co-local collaborative contexts, which would be the preferred way of working according to Scrum and the underlying agile values, as explained in Chapter 3.6.4. Likewise, the limitations also result from a functional overblow, causing the usability problems mentioned in Section 7.2.2.2, which make the systems difficult to work with.

On the other hand, the current NUI-based systems are substantially different since they are typically designed as multi-user interfaces and, therefore, might be a better choice for collaborative environments. However, the current systems are severely limited in features and functionality. As seen in Section 8.5, they only focus on the aspects of sprint planning and operating a basic task board, leaving out many aspects of Scrum. So, in terms of features, the existing NUI approaches only represent limited research prototypes, but they are far from being an applicable working solution for real agile environments.

In summary, neither the existing agile ALM applications nor the current NUI approaches provide a perfect match with the agile philosophy and the special demands of collaborative Scrum teams. While rich in features and technically mature, GUI applications lack support for co-local collaborative activities and suffer from overblown UIs.

On the other hand, touch-based NUIs still show strong technical weaknesses, for instance, in the recognition and processing of touch

points. Moreover, the existing approaches show poor interface designs and inappropriate interaction techniques, and they reveal large gaps on the feature side regarding various aspects of the Scrum framework.

As a novel solution, the next chapter is about combining both paradigms, i.e., classical desktop GUIs and modern touch-based NUI technologies, to an *integrated solution* representing an *interactive Scrum space* carefully designed for single-user as well as collaborative activities of the Scrum framework.

9

INTRODUCING AN INTERACTIVE SCRUM SPACE

Given the landscape of today's touch-based NUIs, it is interesting that they strongly relate to Weiser's vision of ubiquitous computing from 1991, as explained on Page 220. Based on his premise that computers should "fit the human environment, instead of forcing humans to enter theirs" [282], Weiser thought about pervasive displays of different sizes and form factors and envisioned three scales of interaction devices, each optimized for particular tasks.

Inch-scale tabs have been described "as computationally enhanced Post-It notes," *foot-scale pads* were designed as digital sheets of paper, and *yard-scale displays* were envisioned as interactive whiteboards. These devices, however, were not supposed to exist in isolation. As stated by Weiser, "[the] real power [...] emerges from the interaction of all of them" [282]. With this, he shifted the focus away from a single interface towards an *interactive workspace*, where various devices enter the focus of attention when needed and vanish into the background when not. To achieve this, he further envisioned permanent connections between these devices and ubiquitous software applications.

Today, this "computation of the inch, foot, and yard" [77] indeed became a reality. Smartphones (inch-size), tablets (foot-size), and interactive wall displays or tabletops (yard-size) are permanently connected to the internet and thus provide everything needed to create ubiquitous applications for various work scenarios.

This is also recognized by Shneiderman and Plaisant, who verify Weiser's influential idea and believe in the opportunities arising therefrom. According to them, touch-based NUIs, i.e., smartphones, tablets, tabletops, and interactive wall displays, can be integrated to provide more productive work environments [252].

Design considerations of interactive spaces

Such environments are referred to by Haller et al. as "interactive spaces" and are defined as rooms incorporating different digital surfaces in a single space to facilitate work processes [108], e.g., by seamlessly providing data during face-to-face collaboration, thus enhancing social interactions, and allowing users to establish more inventive and creative workflows [131].

Regarding interactive spaces, Haller et al. further identified the following design considerations [108, p. 442].

Multiplicity and heterogeneity of tasks

The first is to consider the *multiplicity and heterogeneity of tasks* arising from the wide variety and diversity of collaborative activities, which they consider one of the main challenges for interactive workspaces. Presentations, for instance, call for different interaction techniques and design considerations than brainstorming sessions and decision-making. Hence, the design of an interactive space should be tailored to a particular use case and also consider that collaborative activities can be composed of various tasks, accommodating individual work phases simultaneously performed by each participant.

Creation and access of shared documents

Furthermore, interactive workspaces should have features that enable the collective *creation and access of shared documents*, as this is a common practice during face-to-face collaboration. According to Haller et al., this promotes the development of mutual understanding, facilitates the coordination of activities, and provides a shared memory for the group. Therefore, during collaboration, it is essential to be able to create and manipulate these documents simultaneously, and they should be viewable by everyone involved at the same time.

Integration of meetings into overarching activities

Moreover, collaboration often involves a larger activity that extends beyond the current session. For instance, a tabletop or board interaction may be just one aspect of a meeting, which, in turn, may be part of a larger project. Hence, it is important to *integrate the current meeting's work into overarching activities* and the larger context by incorporating previously generated information and ensuring that the information produced during the meeting is easily accessible later on.

Combined use of mobile and collaborative interfaces

Lastly, Haller et al. suggest using *combinations of mobile and collaborative interfaces* to exploit the individual benefits of these devices and overcome their shortcomings. For example, when it comes to digital tabletops and wall displays, using software keyboards has been found to be very unpleasant and should be avoided. This presents an opportunity to use mobile devices, such as smartphones, as remote keyboards for collaborative interfaces because users rely on typing on these devices on a daily basis, e.g., when composing short messages.

Mobile devices can also serve as intelligent, general-purpose controllers that allow interaction with interactive tabletops and wall displays beyond simple character input. For instance, they can be used to enhance the creation of new shared artifacts or better interact with existing ones, which has been confirmed by various research studies, e.g., by Roth and Unger [223], McAdam and Brewster [173], or Dachselt and Buchholz [66]. Moreover, since mobile devices are inherently personal and most collaborators are likely to possess one, they can be used as private displays in conjunction with sharable interfaces to facilitate fluent transitions between personal and group activities during collaborative work.

Combined use of tabletops and wall displays

Besides the combination of mobile and shared group interfaces, interactive tabletops and digital wall displays can also be used to complement each other. While tabletops promote collaboration and concurrent multi-user interaction, wall displays are better suited for communal viewing and providing a shared perspective for all collaborators. Therefore, digital boards can augment interactive tabletops by simultaneously displaying contextual information relevant to and readable by all collaborators. Furthermore, as collaborative efforts are commonly composed of group work and phases of presentation, the conjunctive use of these devices also has the potential to enable fluent transitions between these types of activities.

Behind this background, this chapter introduces a novel interactive Scrum space whose purpose is

- to better support Scrum's collaborative activities, i.e., the mandatory Scrum meetings,
- and to overcome the weaknesses of existing touch-based NUI approaches in the context of agile environments.

Furthermore, the system should go beyond the scope of a research prototype. This means it should support the entire Scrum process, not just parts of it, and be technically mature enough to be used in both research environments and industrial practice.

9.1 OVERVIEW

Figure 9.1 shows that the envisioned interactive Scrum space consists of four different interface types, all connected via an agile project management application called *edelsprint*. This web application has been developed to fulfill the design considerations of interactive spaces proposed by Haller et al., as explained in the previous section, and with special attention to

- offer Scrum-specific functionality for all activities and tasks during a sprint cycle,
- address the Scrum challenges and issues that have been identified in Chapter 6,
- overcome the usability issues of existing agile ALM applications that have been analyzed in Chapter 7,
- enhance the Scrum workflow by supporting both individual tasks and collaborative activities,
- and therefore distributing functionality across personal (desktop and mobile) and collaborative interface types (tabletop and interactive display).

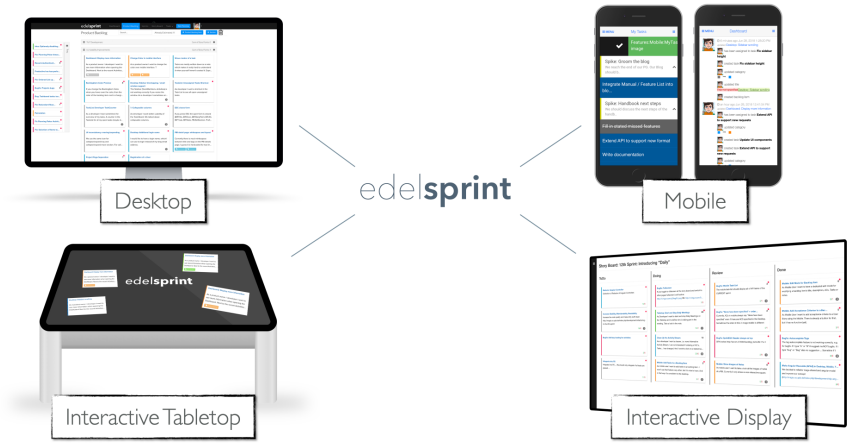


Figure 9.1: edelsprint interfaces

Since edelsprint is a web application, all interfaces can be simultaneously accessed by many clients, i.e., through the browser of each user's end device.

Desktop interface

The *desktop interface* is the primary interface for all personal and private work situations, whereby "desktop" not only refers to stationary desktop PCs but also to mobile laptop computers. With its mouse or touchpad and a hardware keyboard, the desktop interface allows quick and pixel-precise interactions and haptic text input. For the product owner, the desktop interface is used for all tasks relating to managing the product backlog and preparing user stories for succeeding sprints. For the Scrum master, it allows monitoring the sprint progress, preparing meetings, and supporting the development team to achieve the sprint goal. For developers, the desktop interface is used to manage tasks and make the development progress transparent to the rest of the team. Besides these personal work situations, the desktop interface is also used within group settings when plugging a laptop computer into a projector to display its content to the meeting attendees.

Mobile interface

The *mobile interface* is also used for personal work. However, compared to the desktop interface, it only provides a reduced set of features, which benefit from being always accessible and the fact that mobile phones are always ready at hand. Primarily, this set includes features for managing user stories and tasks. The product owner, for example, can use the mobile interface to quickly create drafts of new stories when being at the customer and (perhaps while having a walk) discussing new ideas. Likewise, a developer can use the mobile interface to check off tasks or create new ones while chatting with colleagues during coffee breaks.

In addition to giving users more freedom and access to their work while being on the go, the mobile interface is also used as an authenticator and secondary interaction device while collaboratively working

on the tabletop. The fact that a mobile phone naturally belongs to one person is, for instance, used during the daily Scrum meeting when a developer can lay his phone on the tabletop to identify himself and show his current work status to the other team members.

Tabletop interface

The *tabletop interface* provides dedicated features for short collaborative group work of a sprint cycle, i.e., backlog grooming and the daily Scrum meeting. It is not intended for longer collaborative sessions, such as sprint planning, review, and retrospective events, because due to their longer timespan, these demand a more presentation-like situation with people sitting, looking at the front, and discussing what is shown to the group. Therefore, for these events, the desktop interface is used and usually displayed to all attendees via a laptop computer linked to a projector. In contrast, backlog grooming and the daily Scrum meeting only allow for a limited number of participants and are much shorter. Therefore, they can (and should, see Chapter 4.7.2 and Page 232) be held while standing, which also contributes to more interaction between participants, which in turn can be better supported by an interactive tabletop with its capability to be operated by multiple persons simultaneously.

Vertical wall interface

The *vertical interactive display* or *wall interface* is used as an addition to the tabletop interface and serves as an *information radiator* that compensates for the tabletop's inability to display information in a way that it can be equally seen from all sides. While the tabletop offers a natural feel and fluid interactions that benefit collaborative backlog grooming and the daily Scrum, the vertical display ensures that content is visible to everyone. Thus, the combination of tabletop and vertical display strengthens their benefits while simultaneously compensating for their individual weaknesses, as explained in Chapters 8.4.2 and 8.4.3.

9.2 IMPLEMENTATION

The following two sections explain the overall system architecture and elaborate on the custom-built tabletop hardware.

9.2.1 *System Architecture*

Figure 9.2 illustrates the system's architecture. At the top, it shows three backend components: the *Application Server (API)*, the *Real-time Application Server*, and a *PostgreSQL Database*. At the bottom, it shows the front end represented by single-page applications (SPAs) for the four interface types: desktop, mobile, tabletop, and wall interface.

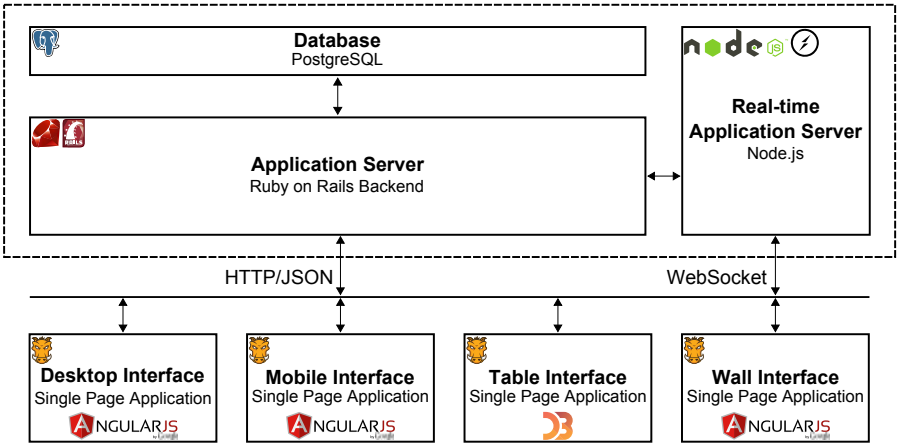


Figure 9.2: edelsprint architecture

The application server is implemented in *Ruby on Rails*[1] and provides the API (Application Programming Interface) for executing CRUD (create, read, update, delete) operations on the PostgreSQL database. This API is used by the four interfaces of the edelsprint software (desktop, mobile, tabletop, wall). It can be called through RESTful URIs, which on the client side are handled by *AngularJS*[2] resources and asynchronous HTTP requests (AJAX).

In addition, a *Node.js*[3]-based real-time application server (also called reactive server) is used for establishing *WebSocket* connections to the interfaces. Because of this advanced technology, it is "possible to open a two-way interactive communication session between the user's browser and a server," which means that the user's browser "can send messages to a server and receive event-driven responses without having to poll the server for a reply" [182].

Therefore, for each API call updating the data model, the real-time server is also triggered and automatically distributes the transferred data to the connected interfaces. As a result, the edelsprint application becomes "reactive" so that changes made by a user are passed to all other users on the fly and without the need to refresh the page, thus overcoming the technical limitations of existing agile ALM applications and the resulting usability problems, as mentioned in Section 7.2.2.5.

Moreover, each interface is implemented as a so-called *single page application* (SPA) built with either the AngularJS (in case of desktop, mobile, and wall interfaces) or the D3[4] framework (in case of the tabletop interface). Due to these SPAs, client-side routing and AJAX allow dynamic rewriting of the current web page with new data from

1 https://rubyonrails.org/
2 https://angularjs.org/
3 https://nodejs.org/en/
4 https://d3js.org/

the server instead of loading new pages entirely. In addition, edelsprint uses content caching and stores user-specific interface states within the local browser cache. These mechanisms allow faster transitions to already loaded pages and make the web application feel more like a native app, e.g., by preserving the scroll position or the collapsed/expanded state of certain UI components.

9.2.2 *MisterT and Object Recognition HOUDINI*

While the desktop, mobile, and wall interface could be realized with standard components, the tabletop was custom-built with particular attention to overcoming the technical weaknesses of existing tabletop systems, as explained in Chapter 8.6.

Especially in academic settings, tabletops usually rely on optical tracking mechanisms and image processing to determine the location of touch points on the surface. That is because bigger tabletops relying on optical tracking can be built at a relatively low cost.

Optical touch tracking

The principle of *optical touch tracking* is shown in Figure 9.3. The user interface gets projected onto a glass or acrylic screen that is equipped with a diffuser foil to "stop" the projected image at the tabletop surface and make it visible to the human eyes. For tracking touch points, the surface is enriched with infrared (IR) light that gets reflected down into an IR camera as soon as fingers or objects hit the surface. Then, the extracted touch information is further processed within image processing software before it is sent to the application, where it is used to trigger user interface events.

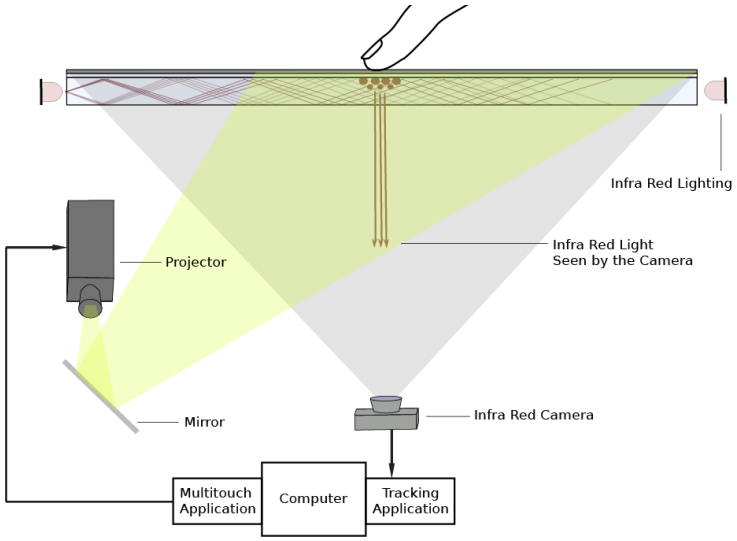


Figure 9.3: Optical touch tracking[5]

5 Source: https://sethsandler.com/multitouch/

The predominant methods for optical touch tracking only differ in how they enrich the touch surface with IR light. They are outlined in the following, including their pros and cons, which are summarized in Table 9.1 to be considered when building a custom tabletop system.

Rear Diffused Illumination

Rear Diffused Illumination (RDI) is one of the earlist optical tracking approaches [171]. As shown in Figure 9.4, the IR illumination occurs from below the tabletop surface, which can be achieved with cheap standard components, e.g. IR lamps.

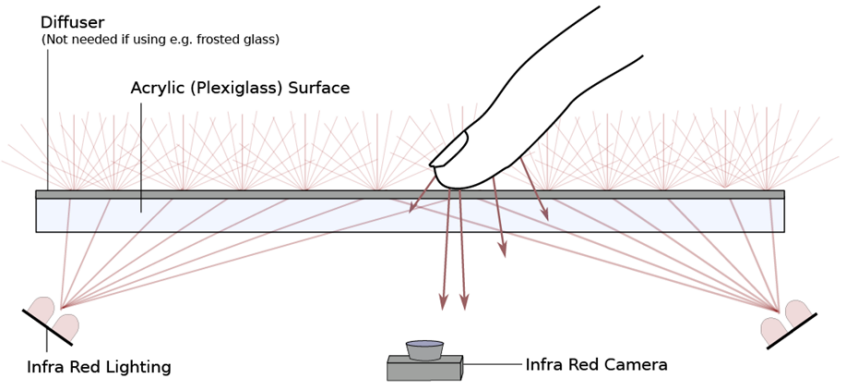


Figure 9.4: Rear Diffused Illumination (RDI)[6]

Object tracking with fiducial markers

Besides touch, this simple setup also allows the tracking of objects through attached *fiducial markers* [136]. As shown in Figure 9.5, a fiducial is a special symbol that can be easily printed on paper and is attached to the bottom of an object, allowing the camera to calculate the object's position and rotation, given that the fiducial (and hence the object's bottom) is well lit with IR light.

Figure 9.5: Fiducial markers [135]

However, the downside of RDI is the relatively low tracking quality, causing false inputs, which can massively affect the user experience. These tracking errors result from the difficulty of illuminating the underside of the tabletop surface evenly (causing so-called tracking hotspots) as well as the fact that Rear-DI causes touch-points with low contrast that are more difficult to track when compared to other principles, as shown in Figure 9.6.

6 Source: https://sethsandler.com/multitouch/reardi/

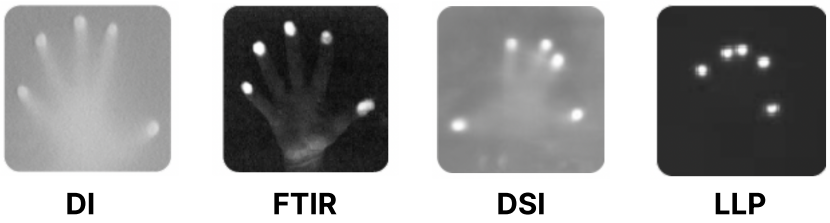


Figure 9.6: Comparison of touch points[7]

Frustrated Total Internal Reflection

In *Frustrated Total Internal Reflection* (FTIR), IR light is brought into the tabletop surface from the side where it is trapped (because of the equally named physical principle) until a touch of a finger makes it reflect down into the camera [111] (see Figure 9.7).

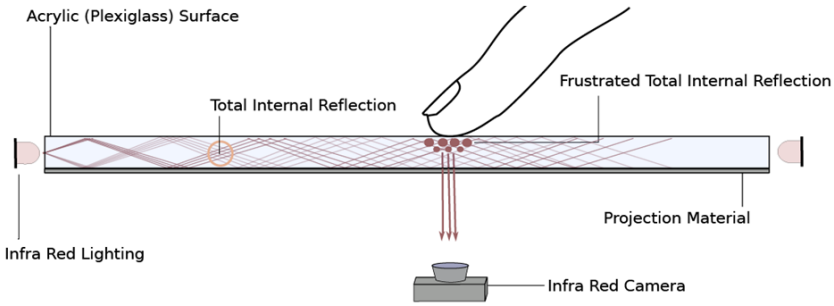


Figure 9.7: Frustrated Total Internal Reflection (FTIR)[8]

FTIR delivers stable touch tracking that allows fast finger movements. That is because of the evenly distributed IR lighting, which delivers high-contrasting touch points. However, FTIR cannot detect objects via fiducial symbols and needs an acrylic surface to ensure the physical effect of total internal reflection. Consequently, larger tabletop sizes are difficult to achieve since acrylic is less stable in shape than glass and deflects with bigger dimensions. Therefore, to counter this problem, the acrylic needs to be considerably thicker, which rapidly increases the overall setup costs.

Diffused Surface Illumination

Diffused Surface Illumination (DSI) has a similar setup to FTIR with IR light coming from the sides, as shown in Figure 9.8. However, it does not rely on the physical effect of total internal reflection but evenly scatters the IR light using a special Endlighten™ acrylic as surface material [2].

Because of this light scattering, DSI allows object recognition through fiducial symbols. However, the Endlighten™ acrylic is rather expensive and shows the same weaknesses regarding form stability as mentioned

7 Source: https://sethsandler.com/multitouch/
8 Source: https://sethsandler.com/multitouch/ftir/

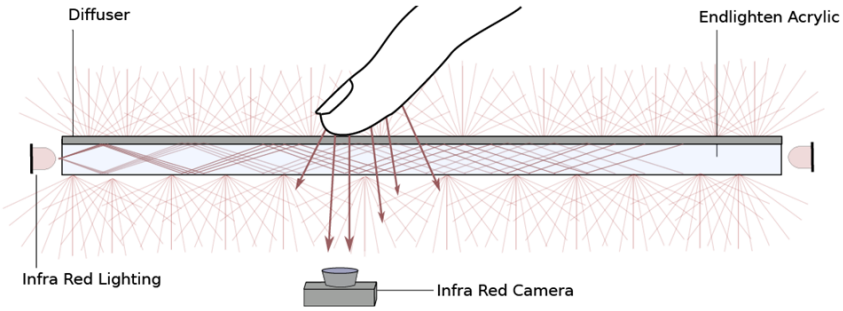


Figure 9.8: Diffused Surface Illumination (DSI)[9]

before. Moreover, the scattering of light also leads to low-contrastive touch points and, therefore, less stable tracking results.

Laser Light Plane

Laser Light Plane (LLP) uses infrared lasers and so-called line-generating lenses to establish a plane of IR light just slightly above the tabletop surface (see Figure 9.9). This light gets scattered down into the IR camera as soon as a finger touches the surface [239].

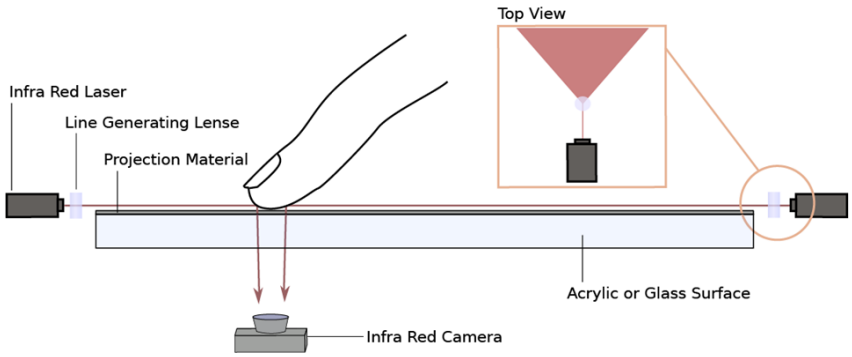


Figure 9.9: Laser Light Plane (LLP)[10]

Since the surface of LLP tabletops is not directly illuminated and the IR camera only captures light when the plane of IR light gets scattered during touch input, the camera image is very rich in contrast, which allows a stable and reliable tracking of touch points, even during fast finger movements. In addition, LLP tabletops can be built with much bigger dimensions. First, this is due to lasers, i.e., the plane of IR light can span an arbitrarily large surface size. This contrasts DI, FTIR, and DSI, where a bigger surface requires more or stronger IR light emitters, which in turn increases the likelihood of IR hotspots leading to bad tracking results. Second, LLP allows using glass panes for the tabletop surface, which results in bigger dimensions because of their stability

9 Source: https://sethsandler.com/multitouch/dsi/
10 Source: https://sethsandler.com/multitouch/llp/

while also being significantly cheaper than acrylic. On the other hand, LLP tabletops lack fiducial object tracking, and their setup is rather complicated because an exact laser alignment is needed to establish an IR light plane just slightly above the surface.

However, both problems could be solved when building a custom LLP tabletop for the interactive Scrum space, which therefor is not subject to the technical weaknesses of other tabletop systems, as explained in Chapter 8.6. Moreover, LLP was chosen since the stability and reliability of tracking touch input, as well as suitable surface dimensions, were considered most important.

	PROS	CONS
RDI	+ cheap components + simple setup + fiducial object tracking	- inequal illumination - low contrast - tracking errors
FTIR	+ equal illumination + high blob contrast + stable tracking	- expensive setup - no fiducial object tracking - limited tabletop size
DSI	+ equal illumination + fiducial object tracking	- expensive setup - limited tabletop size - tracking errors
LLP	+ equal illumination + high blob contrast + stable tracking + large tabletop sizes	- difficult setup - no fiducial object tracking

Table 9.1: Comparison of optical tracking technologies

The built tabletop is called *MisterT*[11] and provides a 160 cm x 90 cm touch surface, which is big enough to fit 6-8 people (the size of a usual Scrum team, see Chapter 4.4.1) comfortably around the sides. For the user interface, it offers 1080p Full-HD resolution provided through rear projection and a special projector foil on top of the glass sheet. The IR laser light plane is established through four lasers with 120° line-generating lenses[12]. Thanks to custom-built mountings at every corner, the lasers can be aligned very precisely so that the light plane can be adjusted to hover as low as 1 mm above the tabletop surface without bouncing off at any position.

11 **M**ultitouch **I**nteractive **S**urface & **T**angible **E**xploration **R**esearch **T**able

12 Using multiple lasers helps to prevent occlusion, which may occur if only one laser is used and when fingers are in alignment so that the front finger may block the laser beam for the rear fingers.

HOUDINI

In addition to the tracking of touch input, MisterT is also equipped with the *HOUDINI* tracking engine, which was developed and published as part of this thesis and represented the first ever method for tracking objects and pen input on LLP-based tabletops [120].

Object tracking

HOUDINI is designed to support fast and reliable tracking of multiple passive objects (without additional electronics). The tracking data includes translation and rotation of objects without limiting the processing speed of touch input data from fast-moving fingers. Additionally, the system supports state changes of objects, i.e., objects can be equipped with a physical button triggering a state change, for example, to confirm a prior selection in the graphical user interface (see Figure 9.10).

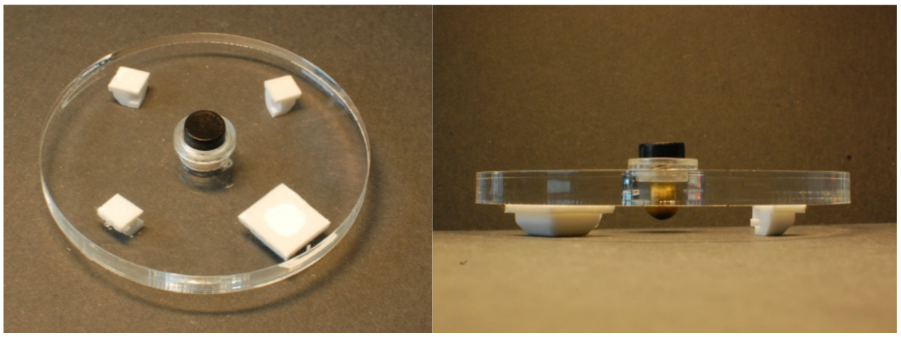


Figure 9.10: Tangible object with attached markers

In contrast to other object tracking approaches, HOUDINI does not rely on visual fiducial symbols but instead uses physical markers attached to the bottom of objects, which break the laser light plane and thus form individual "touch" patterns that the system can identify (see Figure 9.10 and the purple circles in Figure 9.11).

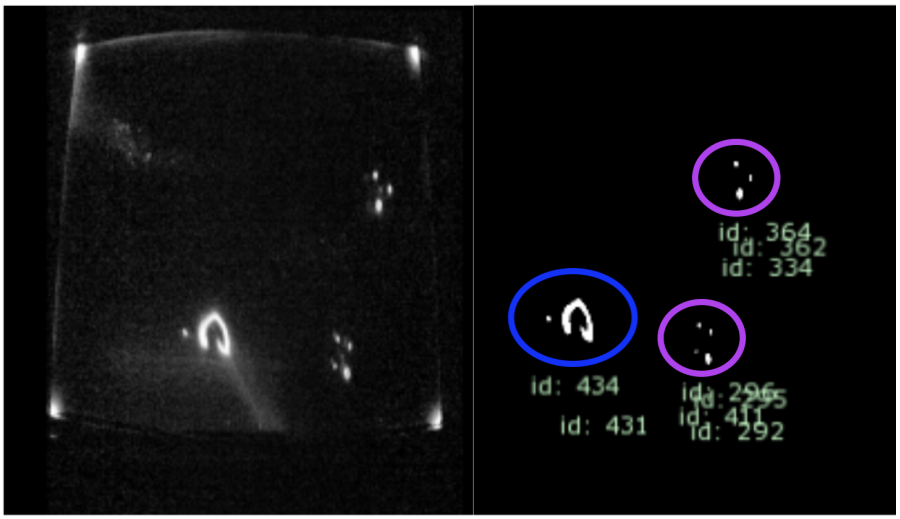


Figure 9.11: Source image (left) and identified pen and object patterns (right)

The massive benefit of this approach is that object recognition is performed at the level of touch information rather than analyzing the camera's video stream frame by frame, resulting in reliable tracking, even when objects are manipulated very fast.

Pen tracking

The principle of recognizing distinct patterns from different touch point arrangements also applies to identifying pen input. Usually, the palm naturally rests on the writing surface while writing with a pen, as shown in Figure 9.12.

Figure 9.12: Resting palm while writing

With the palm breaking the laser light plane, HOUDINI can identify the resulting big touch point as a resting palm. An occurring small touch point located in a circular area with a certain distance to this identified palm can then be interpreted as the tip of a pen (see the blue circle in Figure 9.11).

A particular advantage of this method is that it does not require any additional electronic parts and allows using standard office pens[13], which can be reliably tracked without the need for a special writing surface overlay, which would considerably reduce the contrast of the visual display and is, for instance, required for other pen recognition systems, such as the digital "Anoto Pen" [8].

9.3 FEATURES AND SOLUTIONS TO THE IDENTIFIED CHALLENGES

According to its system design, the edelsprint application offers distributed functionality across personal (desktop and mobile) and collaborative interface types (tabletop and wall display). However, the implemented features are not equally available in every interface. Instead, they are only implemented and accessible where it makes sense according to the underlying task.

13 Pens should be used without removing the cap to prevent them from really writing on the surface.

The following subchapters describe the implemented features and how they relate to the identified Scrum challenges of Chapter 6 and the problems of existing ALM applications, as described in Chapter 7. For better comprehension, they are presented in an order following the sprint logic of Scrum, i.e., beginning with managing the backlog, followed by planning a sprint to executing the Scrum ceremonies.

9.3.1 *The Backlog*

As explained in Chapter 4.4.2, managing the backlog is the sole responsibility of the product owner. It is not a collaborative task and is therefore only implemented for the *desktop* and *mobile* interfaces.

To comprehend the issue of insufficient backlog ordering (see Section 6.2.4.2), edelsprint introduces *backlog sections* as a novel and flexible way to order and manage different parts of the backlog.

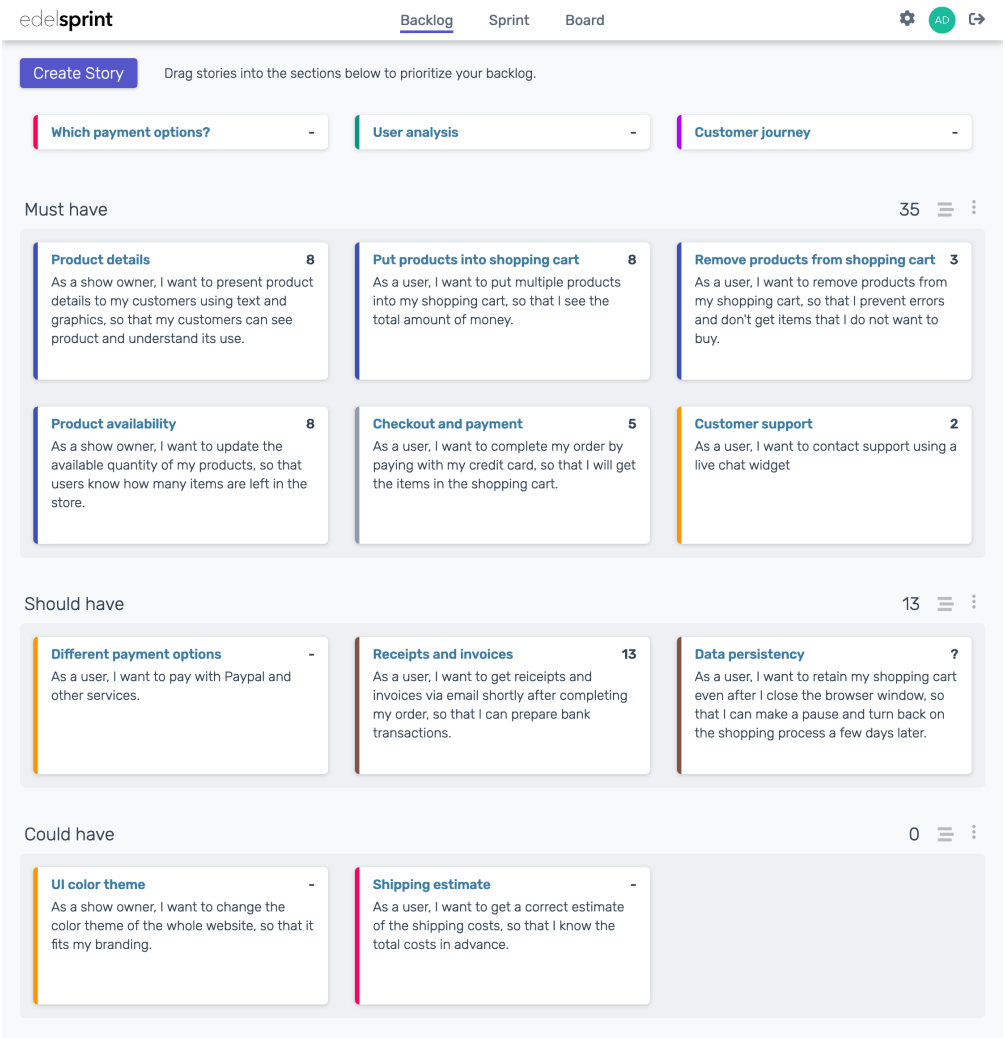


Figure 9.13: The edelsprint backlog

Backlog sections can be used to group backlog items according to freely selected criteria. As an example, Figure 9.13 shows "Must have," "Could have," and "Should have" sections representing the *MoSCoW* prioritization method (see Page 63). Another example might be a "Ready for Sprint Planning" section that could contain all items sufficiently specified to be discussed in the next sprint planning event. Similarly, a section could contain items of a particular category, e.g., all "Frontend" items, or follow different semantics, such as "Things to discuss," or "Top priority."

In addition, sections show the sum of story points contained, which further helps the product owner to prepare future sprints. Moreover, each section can be collapsed (see Figure 9.14), which is useful to save some display space and shift the focus of attention to a particular set of items, e.g., while conducting a sprint planning event.

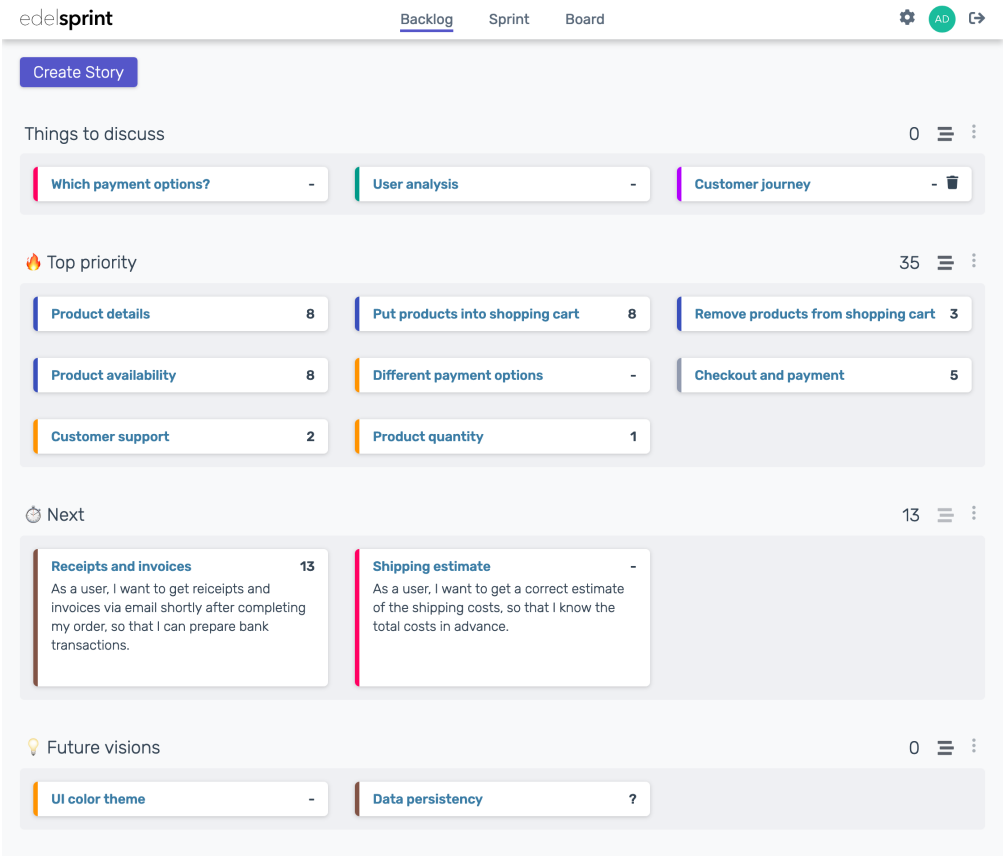


Figure 9.14: Backlog with collapsed sections

While all Scrum team members can create new stories, which will show at the top of the backlog, the product owner is in control of the *ordering*. Items can not only be moved back and forth between sections via drag and drop, but entire sections with their items contained can also be quickly and easily moved further up or down in the backlog.

This arrangement of items and sections should better support the order of the backlog and bring it more into the foreground since it reflects the items' prioritization and, thus, their importance according to the Scrum rules. An additional priority field at the item level is deliberately omitted to counteract contradictions in the prioritization, as they can occur in existing agile ALM applications (see Section 7.2.2.3).

Speaking of backlog items, these are consistently designed to the de facto standard of user stories. As shown in Figure 9.15, a user story is represented as a *card* that contains a title, a description, and a category. In addition, the card also shows the estimated effort in the form of story points (see Section 4.8.2.1) and the person in charge of the story. However, the latter is not necessarily directly involved in the implementation. Rather, he or she should serve as the first point of contact for the whole team in case of questions and assume a certain degree of accountability during sprint execution, e.g., for presenting the story during the sprint review.

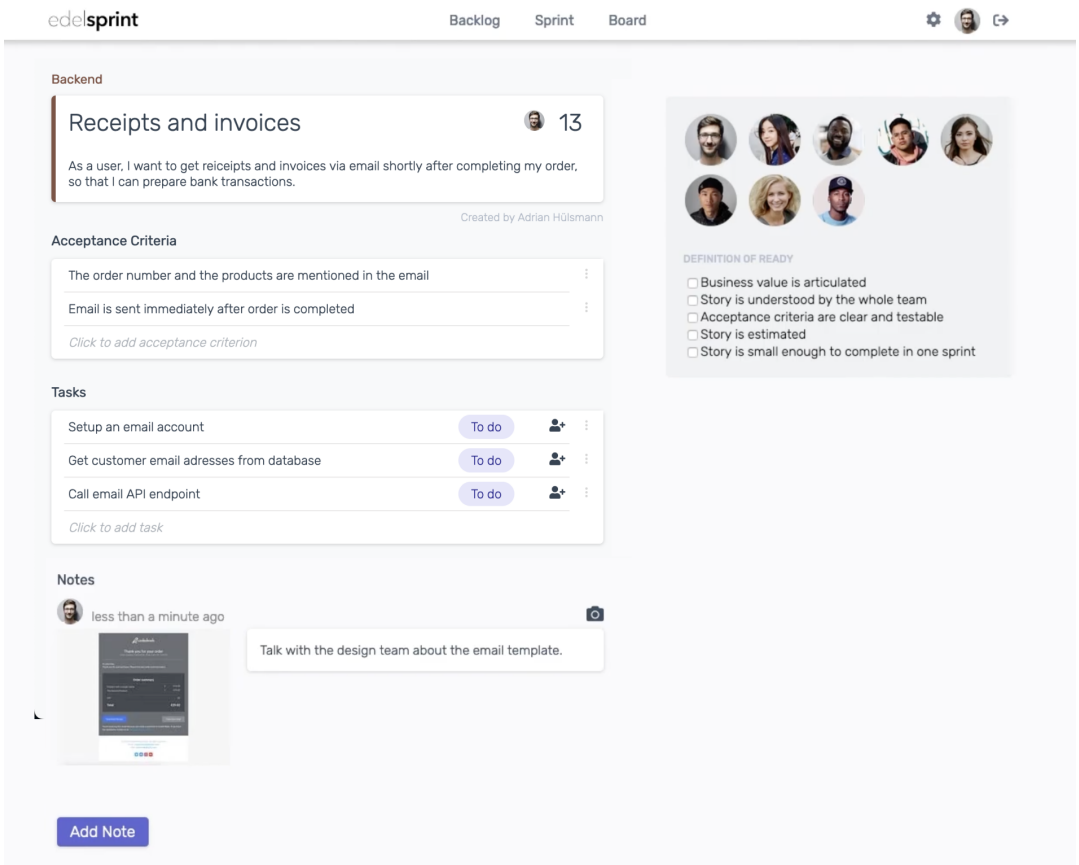


Figure 9.15: User story

Below the user story card, there are three more sections. In the first one, a *list of acceptance criteria* can precisely specify what the user story is about. These acceptance criteria allow the product owner to prepare

the sprint planning and the development team to better plan and test the implementation (see Section 6.2.4.5), which has been lacking in all the investigated applications (see Section 7.2.2.3).

Moreover, since a bad preparation of stories is a common issue (see Section 6.2.4.3), a user story also shows a *definition of ready* located below the Scrum team members at the right side. It is represented as a ticking list with criteria to fulfill before a story is considered "ready" for sprint planning. Thereby, the state of readiness, i.e., the checkmarks, are individually stored for each story and not on a global level. Therefore, the degree of specification of a story can be transparently traced at any time, and the extent to which the exact requirements have been understood by the team, thus counteracting later undesirable developments from the very beginning.

How a story is implemented by the development team, i.e., the exact steps during sprint execution, on the other hand, is mapped in a *list of tasks*. It is located directly below the acceptance criteria to better relate development tasks to the final criteria for fulfilling the implementation. As in other agile ALM applications, task descriptions, the state of progress, and the assignments of tasks to users are visible to the entire team. However, unlike other applications, edelsprint intentionally does not provide options for tracking the time spent on tasks and omits estimating tasks. This is to bring the focus to estimating user stories (and not tasks) and counter the problems from monitoring working hours, as explained in Section 6.2.2.7. Moreover, a task can be assigned to not only one but multiple team members. This should encourage team members to help others (see Section 6.2.6.5), since their help becomes visible to the entire team.

Below the list of tasks, *notes* can be used to add further details to the story, for example, by uploading design documents or image files. Especially during personal meetings, teams often produce quick drafts or whiteboard sketches. Hence, the mobile interface offers a quick upload feature using the smartphone's built-in camera. This way, edelsprint also targets the challenge of proper knowledge management (see Section 6.2.3) by storing all relevant information directly within the user story and making decisions transparent to the whole team (see Section 6.2.3.1).

—

Demo video of managing the backlog:
https://youtu.be/3b4hLT19lFE?si=T3c321lFJDIuzHiy

Demo video of creating a user story:
https://youtu.be/ic3k6LIYA2o?si=hX15p_qvR3_WgMA5

9.3.2 *Sprint Planning*

Since sprint planning can take several hours and involves the entire Scrum team, it is usually conducted in a typical meeting room environment where the product owner presents the sprint goal and explains the prepared user stories.

For this purpose, the *desktop interface* is used and shown to the audience via a beamer. For the execution of the meeting, edelsprint provides a dedicated sprint planning *event* with features carefully designed so that the displayed information and controls can be easily read on a projector, even from a greater distance. This is to prevent less willingness to participate and a decrease in attention among participants, which have been observed with the investigated agile ALM applications, as explained in Section 7.2.2.4.

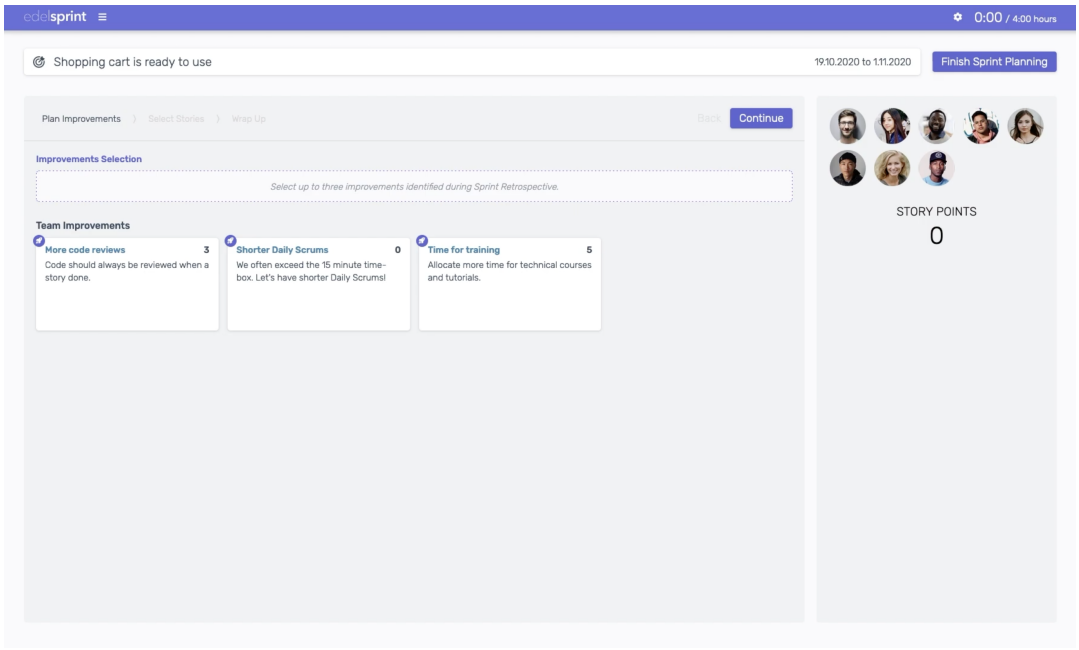


Figure 9.16: The "Plan Improvements" step of sprint planning

As shown in Figure 9.16, the user interface of the sprint planning event consists of four parts. At the very top, the *event bar* replaces the standard menu bar, thus indicating that a meeting is taking place. In order to encourage the attendees to stay focused, it also visualizes the past and remaining time according to the corresponding time box.

Below, the *sprint goal* is displayed in a particular large manner. It is designed to be specified by the product owner before the actual meeting when creating a new sprint and continuously shows during any of the Scrum meetings. All of this is to prevent the problem of teams not using a sprint goal at all (see Section 6.2.5.6) and to address

the issues of existing agile ALM applications, in which the sprint goal is given little to no importance, as explained in Section 7.2.2.4.

Below the sprint goal are the *meeting guide* at the left and the *meeting sidebar* at the right, which will be explained in the following.

The meeting guide helps to process the sprint planning event in a step-by-step manner. At first, the *Plan Improvements* step visualizes the *improvement backlog*, which contains mutual agreements of things to improve in terms of the Scrum process (see Figure 9.16). This step is designed to promote including at least one improvement as an outcome of preceding sprint retrospective meetings since neglecting to work on identified improvements has been identified to be a common issue (see Section 6.2.5.9). In order to emphasize the importance of improvements, they were designed analogously to user stories, i.e., they also contain an effort estimate in the form of story points and lists of acceptance criteria and tasks. This should ensure that improvements are given the same importance as user stories in sprint planning and that their workload must be planned accordingly.

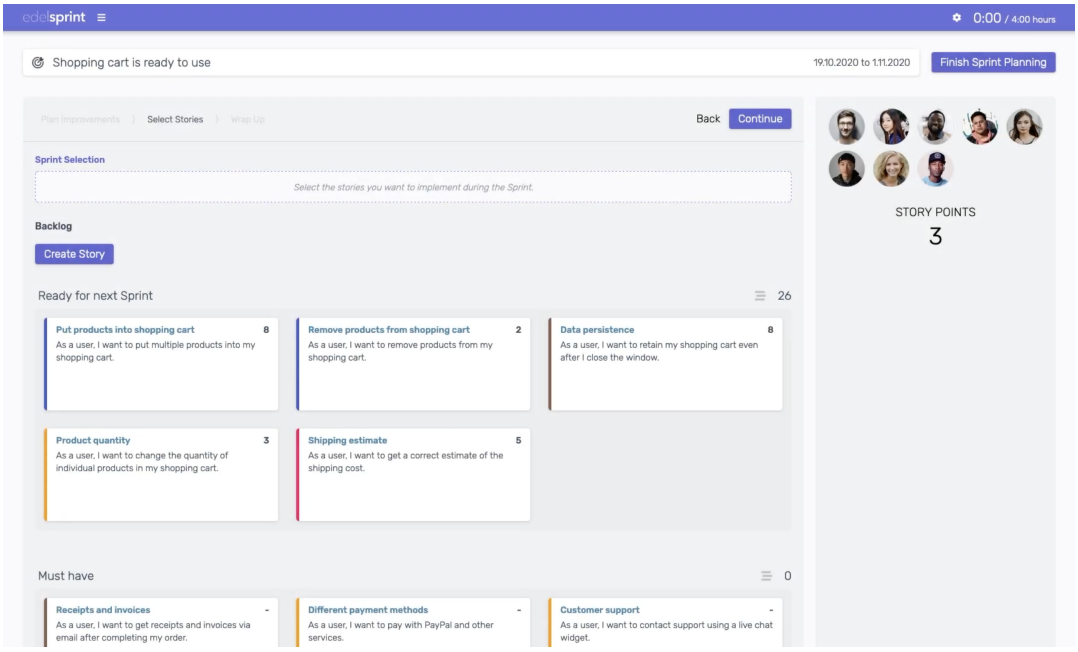


Figure 9.17: The "Select Stories" step of sprint planning

Likewise, the *Select Stories* step of the meeting guide visualizes the backlog with all its sections and stories contained (see Figure 9.17). In both steps, items can be added to the sprint by dragging them into the *sprint backlog section* sitting on top of both backlogs. Due to edelsprint's real-time application server (see Section 9.2.1), any change in data is immediately shown in the UI and does not require manual page reloads, which were found to be disturbing in the investigated agile ALM applications (see Section 7.2.2.4).

The design of the sprint backlog section has been particularly chosen to encourage product owners to prioritize the backlog correctly, i.e., by keeping the most important items at the top of the backlog. By doing so, the dragging distance towards the sprint backlog section is short. If the backlog, however, is not prioritized correctly, this distance will become increasingly larger, thus making the selection of items to add to the sprint consciously more difficult.

Moreover, the sprint planning meeting also provides an estimation functionality for improvements and user stories. For this, edelsprint offers interactive *planning poker* sessions, which is the de-facto standard for agile estimation (see Section 4.8.2.2). During an estimation session, the attendees' mobile user interfaces will automatically update and show the planning poker cards with story points to choose from. After a selection has been made, the desktop interface automatically updates and shows a green checkmark above the person's avatar to indicate that he/she has given a vote. Once all participants have submitted an estimate, the session is automatically closed, and the results are displayed in a *distribution graph* (see Figure 9.18).

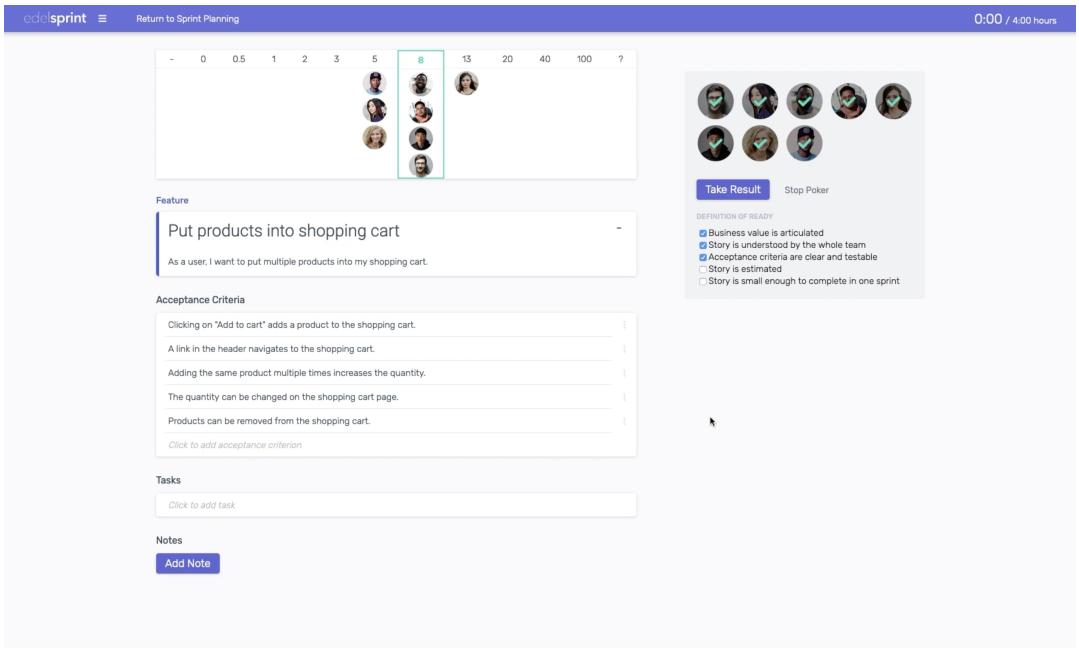


Figure 9.18: Planning poker results

While edelsprint highlights the most picked story point value, the attendees can either decide to take it or further discuss the results in case of uncertainties and start another session. To reduce the cognitive load, which has been a problem of the planning poker implementations of the investigated agile ALM applications (see Section 7.2.2.4), all of the story details remain visible during the estimation session. In addition, the desktop interface also randomly shows two already estimated stories, which serve as a reference and thereby facilitate the

current estimation. Moreover, the interactive session has been designed to still support individual navigation, which means that users can also navigate to other stories and make comparisons on their mobile phones without affecting the flow of other attendees.

Furthermore, the planning poker feature has been designed to facilitate team-based decisions and address the problem of dominant product owners (see Section 6.2.5.10). Several things should contribute to this. On the one hand, the participation of the product owner in the voting can be switched off so that he cannot influence the result himself. On the other hand, the product owner is also made more responsible in the preparation since the voting is carried out exclusively at the level of the acceptance criteria, which must be prepared accordingly. As mentioned before, effort estimation on the task level is not possible, which should avoid unnecessary and time-consuming discussions. This design decision and the clear structure of user stories are also intended to do better justice to agile planning and counter the problems resulting from too-detailed planning (see Section 6.2.5.5).

Next to the meeting guide is the meeting sidebar, which shows the attending and absent Scrum team members. For the sprint planning event, it also prominently shows the *definition of ready* as a tick list for each user story, thus targeting the problem of teams ignoring the definition of ready (see Section 6.2.5.7). In addition, the meeting sidebar shows the sum of story points of backlog items and improvements that have been selected for the sprint. This shall further contribute to treating sprint planning as a mutual agreement between the product owner and the development team regarding the sprint scope, while the definition of ready ensures that the development team has everything needed to transform requirements into a working product increment.

Regarding the sprint scope, the problem of continuous over-estimation (see Section 6.2.5.1) is also addressed by saving the actually implemented story points, i.e., story points of "done" user stories, at the end of a sprint and making them available in the *sprint history* (see Figure 9.19). This enables the determination of a team's velocity and, thus, an understanding of the usual work performance of the team, which can be taken into account in advance during the sprint planning event.

The sprint history is also used for knowledge management (see Section 6.2.3) and should lead to a better understanding and traceability of previous decisions (see Section 6.2.3.1) through the documentation of past sprint goals and implemented stories, which are available at any time and can be opened for each of the past sprints.

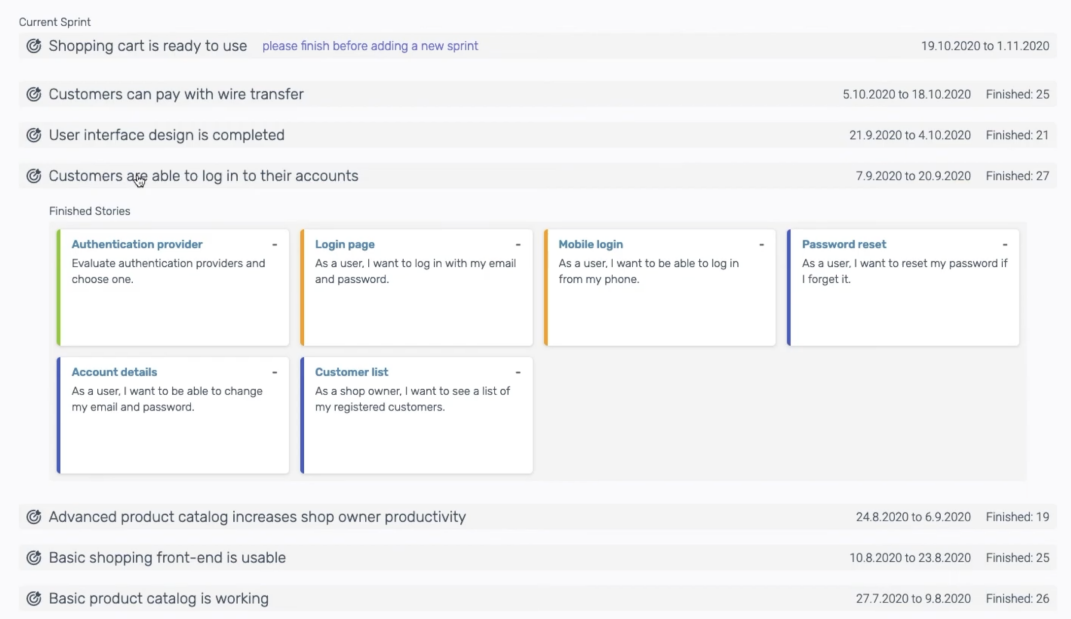


Figure 9.19: The sprint history shows all previous sprints including the "done" stories

—

Demo video of sprint planning
https://youtu.be/v7t5_0yLLcY?si=XIAJtVpUBWbC-Syw

Demo video of creating a sprint:
https://youtu.be/Bt-6KTYbn6A?si=kcms3duXpuVoJ4zS

9.3.3 *Sprinting and Daily Scrum*

During a sprint, the development team transforms the user stories of the sprint backlog into an executable product increment. For this purpose, developers specify *tasks* describing the necessary steps to fulfill the user stories' acceptance criteria. To emphasize this relationship in edelsprint, tasks are managed directly under the acceptance criteria of a user story.

A single task consists of a short description, a status, and an assignment to one or more developers (see Figure 9.20). The progress of the task can be mapped via the status, which can take one of four values. *To do* means that the task has been defined but has not started. *In Progress* means that the task is currently being processed by the person responsible. With *Blocked*, a developer can express that the work on this task is blocked and cannot be continued. The completion of a task is represented by the *Done* status.

Tasks can be defined and updated via both the desktop and the mobile interface. While the desktop interface allows developers to

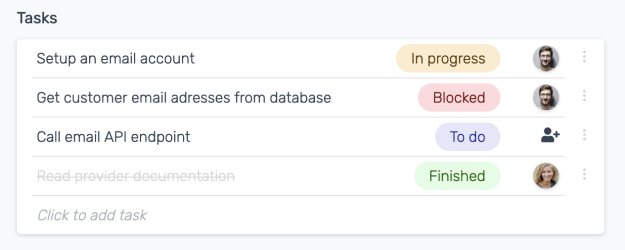


Figure 9.20: Tasks

work on the same device and quickly switch back and forth between the familiar programming environment and their own task management, the mobile interface, on the other hand, offers the advantage of being able to quickly manage tasks on the go. This enables better digital support for ad-hoc meetings, such as those between developers having a chat at the coffee machine, by allowing content and outcomes of spontaneous discussions to be captured directly. This includes the ability to quickly add notes and images to a user story using the mobile interface and smartphone camera.

Furthermore, in contrast to the agile ALM applications examined, edelsprint also offers the possibility to work with a *definition of done* and thus improve control over the technical debt during a sprint (see Section 7.2.2.5). The criteria for fulfilling the DOD can be defined individually for each team in the settings. Once a story is added to the sprint, it no longer shows the definition of ready but the definition of done instead. Like the DOR, the DOD is then displayed as a checklist on the detail page of a user story under the team (see Figure 9.21). The status of the criteria, i.e., whether they have been met or not, is stored individually for each user story, i.e., the developers can manage the degree of fulfillment of the DOD for each user story and thus track the measures to reduce technical debt in the course of a sprint.

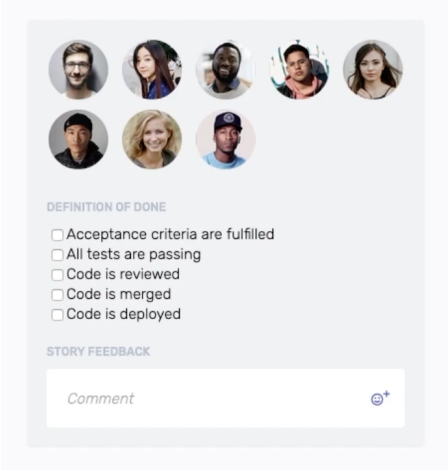


Figure 9.21: Definition of done

Besides working on tasks and updating the definition of done of particular stories, the team should also have a permanent understanding of the overall sprint progress, thus making it possible to adapt their work accordingly. For this reason, edelsprint provides the *board*, which is a specially designed task board consisting of four columns, *To do*, *In Progress*, *Review*, and *Done*, which reflect the work status of the user stories contained (see Figure 9.22).

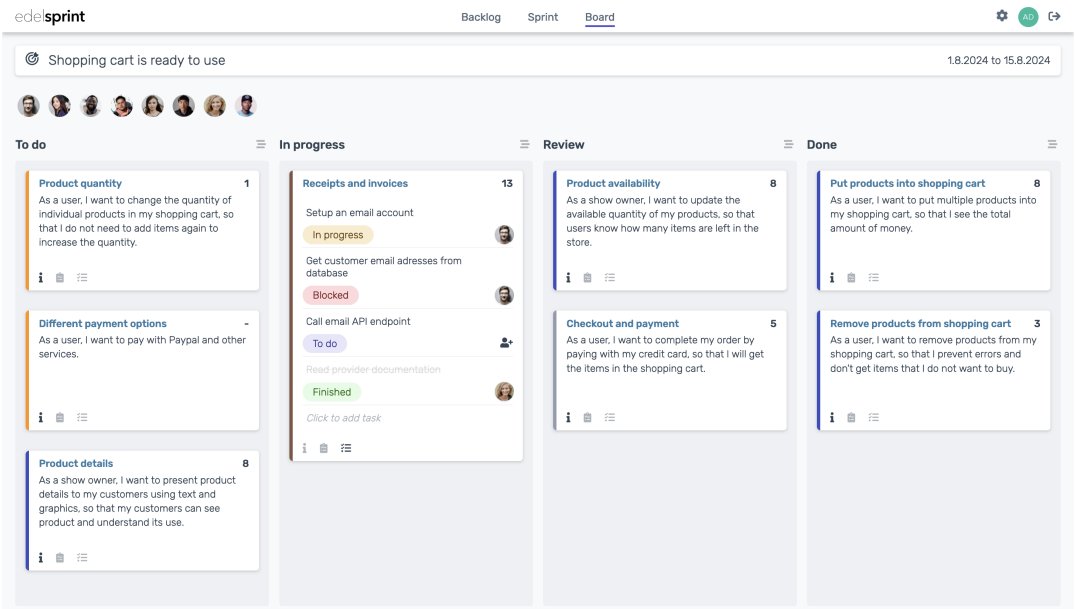


Figure 9.22: Board

At the beginning of the sprint and right after the sprint planning has been finished, all selected user stories (and improvements) appear in the "To do" column. During the sprint, user stories can be moved through the columns via drag and drop, thus making it possible for the whole team to understand the sprint progress.

While this design is very common and was found in all other investigated agile ALM applications, the edelsprint board offers a far more advanced feature set. First, it shows the sprint goal prominently on top, thus serving as a constant reminder of what to achieve during the sprint so that the team can base decisions accordingly. Second, it shows the Scrum team members, which can be clicked to filter the board for user stories and tasks to which the selected person is assigned.

Moreover, the board allows users to view all story details separately and independently of all other user stories. For this, the card's content can be switched to either show the story's description, acceptance criteria, or tasks, as shown in Figure 9.22. The benefit of this design is that developers can look at the details of individual user stories, e.g., the ones they currently work on, while simultaneously keeping an overview of all other user stories without being visually distracted.

For the coordination of developers among each other, edelsprint provides an interactive daily Scrum, which has been designed as a standup meeting carried out at the tabletop in combination with an interactive vertical display. This particular setup was chosen because tabletop and vertical display complement each other well, thus compensating for their respective weaknesses, which have been discussed in Sections 8.4.2 and 8.4.3.

The vertical display is an information radiator for the team and provides two views that can be switched with a button. It either shows the task board mentioned before or the *burndown graph* of the current sprint, both styled to be operated via touch and comfortably read from a further distance. During a daily Scrum, the display also shows a 15-minute timer, which shall help the attendees to keep the meeting short, thus addressing the common problem of daily Scrums exceeding the timebox, as explained in Section 6.2.6.4.

While the vertical display's primary purpose is to display information to all attendees of the daily Scrum, the tabletop serves as an interaction device for the currently speaking person. Due to the edelsprint architecture (see Section 9.2.1), triggered functionalities at the tabletop automatically update the UI of the vertical display, thus allowing the attendees to follow along without being disturbed by a blocked view.

By default, the tabletop shows the user stories of the current sprint in a card-like manner, which can be arranged freely on the surface. When tapping a user story, it is brought into focus, and both the tabletop and vertical display show all of its details, including acceptance criteria, notes, and tasks. The latter are also draggable components on the tabletop that arrange themselves around the user story they belong to.

Moreover, edelsprint is capable of facilitating the daily Scrum meeting by supporting developers to answer the typical three questions: "What did I do yesterday?", "What will I do today?" and "What are my impediments?" (see Section 4.7.2). For this, the tabletop identifies the current speaker via his or her mobile phone laid on the tabletop surface and tracked via the custom device recognition HOUDINI (see Page 264).

When a user is identified at the tabletop, the task board on the vertical display is automatically filtered, i.e., it only shows the stories the user is involved in and displays the corresponding task lists. In addition, tasks the user has been working on since the last daily Scrum are automatically highlighted, thus reducing the cognitive load to recall "What did I do yesterday?". To answer the question "What will I do today?", the user can assign himself to tasks by dragging them from the tabletop surface towards his mobile phone. As a result, the task board automatically updates and highlights these new task

assignments. Finally, for answering "What are my impediments?" the task board also highlights tasks that are in the "Blocked" status, again making it easy to recall problems and ask others for help.

Altogether, these features are intended to keep the attention high, thus targeting the problems mentioned in Sections 6.2.6.3 and 6.2.6.2 and should help to increase the overall team spirit by keeping the meeting short and engaging.

—

Demo video of the board:
https://youtu.be/KODsIf7e2QI?si=QgdO4KaDrhGbnCch

9.3.4 *Backlog Grooming*

Grooming meetings are a proven means of refining the backlog as preparation for the next sprint planning event and thus represent an important connection between the product owner and the development team, as described in Chapter 7.2.2.7.

For the product owner, the advantage of the grooming meeting is that technical aspects of the implementation can already be considered during the planning and prioritization of backlog items. For example, the development team can uncover technical dependencies between user stories that are difficult for the product owner to assess alone. The advantage of the grooming meeting for the developers is that they can better prepare for the content of future sprints and, thus, if necessary, make appropriate preparations that facilitate their development work.

However, as seen in Chapter 6.2.4, conducting grooming meetings with members of the development team is a challenging task for the product owner that is often neglected (see Section 6.2.4.4). Moreover, as shown in Chapter 7.2.2.7, none of the investigated agile ALM applications offers dedicated features for supporting grooming meetings.

In the edelsprint system, grooming meetings take place at the tabletop (see Figure 9.23). A meeting can be initiated on the product owner's mobile phone when it is laid on the tabletop surface. Once the meeting starts, the mobile interface automatically switches to the backlog view, displaying all items in a scrollable list. Since grooming meetings only focus on a few backlog items (usually the ones to be considered for the next sprint planning event), the product owner can decide which items to discuss by swiping these items from the backlog of the mobile phone onto the tabletop surface.

Due to the edelsprint architecture (see Chapter 9.2.1) and the HOUDINI object tracking system, selected backlog items will appear immediately on the tabletop surface and automatically arrange around

Figure 9.23: Backlog grooming at the tabletop

the mobile phone's position, thus providing a smooth user experience that blends interactions between the mobile and tabletop interface.

On the tabletop surface, backlog items can be dragged around, resized, rotated, and focused as desired. This *free spatial arrangement* of backlog items allows the product owner to prioritize stories ad hoc during the discussion by visualizing dependencies through different spatial arrangements, e.g., by making use of the law of proximity (see Page 187).

When an item is being discussed and brought into focus by double-tapping on it, the card animates to a bigger version, providing a predefined set of *quick tags* (see Figure 9.24).

Depending on the discussion result, an item can be easily tagged as:

- **Delete** - to indicate that the item is no longer needed,
- **Discuss** - to indicate that this item has been discussed, but it still needs more elaboration to investigate its requirements,
- **Refine** - to mark the item as being too complex, so it should better be split into multiple backlog items,
- **Ready** - to indicate that the item can stay as is, i.e., from the developers' perspective, it is ready to be presented during sprint planning.

Besides supporting a collaborative grooming meeting at the tabletop, the developed system was also designed to bring the meeting results into action on the product owner's side. For this, all backlog items processed during a grooming meeting are transferred into a new section within the product backlog, whose title refers to the grooming meeting and the time it took place. This feature is intended to help the

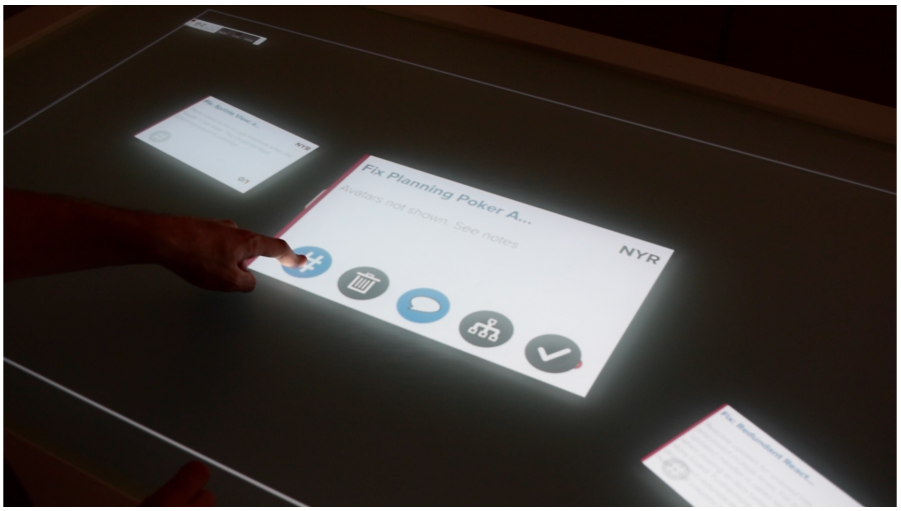


Figure 9.24: Quick tags

product owner by reducing the cognitive load for recalling discussed items when returning to his desk so that he can immediately begin to refine the backlog according to the outcomes of the grooming meeting.

9.3.5 *Sprint Review*

The sprint review aims to gather feedback about the developed product increment, then initiate a revision of the backlog and consider the new findings for further product development. It is, therefore, an integral part of Scrum's inspect and adapt approach as part of the built-in empirical process control (see Page 75). However, as described in Section 6.2.7.4, the common problem is that the sprint review is often treated as the venue of approval, which can result in unnecessary discussions in the presence of stakeholders and thus massively restrict the flow of feedback. To work against this problem, edelsprint pursues two approaches. On the one hand, functions are offered to review user stories already during the sprint. On the other hand, it provides dedicated support for core aspects of the review meeting.

The establishment of a review process in the course of a sprint begins with preventing subsequent modifications of the sprint backlog to overcome the review issues mentioned in Section 6.2.4.6. edelsprint has a built-in role model that prevents developers from modifying the sprint goal and adding or removing items from the sprint backlog, thus ensuring the sprint's integrity as planned. Subsequent modifications are limited to the product owner role, but as explained in Section 4.7.1, these shall be an absolute exception, which the application also draws special attention to by warning the product owner when modifying an ongoing sprint.

In order to establish a tight feedback loop between the development team and the product owner, edelsprint furthermore provides *review requests*. When working on a user story, a developer might want feedback from the product owner regarding the implemented solution. For this, edelsprint provides a "request review" button below the list of acceptance criteria. When triggered, an indicator shows that the user story is currently being reviewed by the product owner. Simultaneously, the story shows in the "Review requested" section of the product owner's *review list*. This list consists of four sections containing all user stories of the sprint. From top to bottom, these sections are:

- Review requested: user stories with active review requests
- Partially accepted: user stories where some but not all acceptance criteria are met
- Accepted: user stories for which all acceptance criteria are met
- Not reviewed: user stories for which a review has not yet been initiated

When a story shows under "Review requested," the product owner can review its implementation and mark individual acceptance criteria as being fulfilled or not. If the story meets the product owner's expectation completely, i.e., all acceptance criteria are marked fulfilled, it is automatically moved to the "Accepted" section of the review list. If only some acceptance criteria are fulfilled, it will show in the "Partially accepted" section instead.

For the development team, which does not have access to the review list, the review results are shown on the user story details page in the form of green check marks (fulfilled) or red warnings (not fulfilled) next to the acceptance criteria. This way, both the development team and product owner have a mutual understanding of the development progress and state of ongoing sprint review.

Another feature contributing to establishing a review process during a sprint is the *definition of done*, which has already been mentioned in Chapter 9.3.3. This checklist with individually checkable criteria is displayed on each element of the sprint backlog and should establish a uniform understanding of which criteria must be fulfilled to consider a sprint backlog item as finished. The display is deliberately chosen to be very present so that, on the one hand, the development team is urged to adhere to the definition of done during implementation. On the other hand, the stakeholders can have more confidence in the quality of the presented product increment because of clarity in communication [152] and increased trust [226].

Besides supporting ongoing reviews during the sprint, edelsprint also provides a dedicated *sprint review meeting*. Similar to the sprint planning, the sprint review has been implemented for the desktop interface and, hence, is intended to be conducted using a projector to provide a good view for all participants.

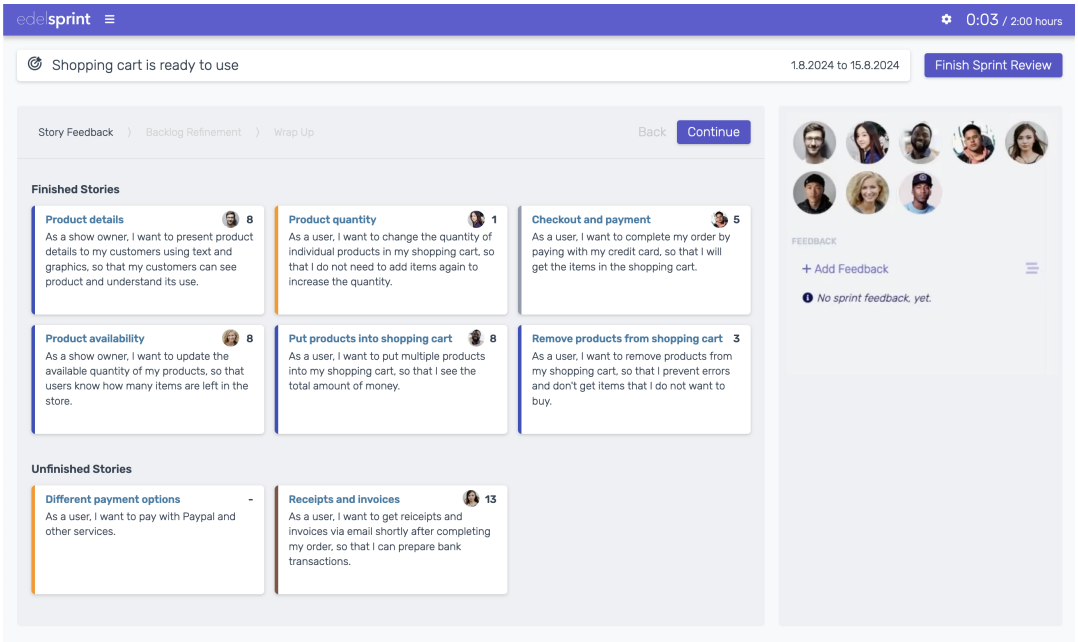


Figure 9.25: The "Story Feedback" step of the sprint review

The main view of the sprint review event is shown in Figure 9.25. It consists of three sections. The first is the *sprint goal*, prominently shown at the top. Below are the *guided meeting tour* and the *meeting sidebar* at the right.

At the beginning of the event, the guided meeting tour starts with the *Story Feedback* step showing all items of the sprint with a clear distinction between finished and unfinished items. "Finished" in this case means that the item has been moved towards the "Done" column of the board and that all criteria of the definition of done are fulfilled. This shall ensure that only finished stories are presented during the sprint review meeting, thus preventing the problems mentioned in Section 6.2.7.3.

In order to establish responsibility for the implemented solution, each user story features a *representative* shown on the top right corner of the card, which can be assigned via the user story's details page. During the sprint, this person is meant to be the first point of contact for the product owner and other Scrum team members in case of questions. This does not necessarily mean that this person is actively involved in working on the story's tasks. Rather, this feature is designed to ensure that a particular person is in charge of keeping the

story's progress up to date and that its result will be presented during the sprint review.

During the presentation, story details, like acceptance criteria, tasks, and notes, can be accessed at all times by clicking on the corresponding card, which will open up the user story details page. This is also true for unfinished stories, which are not completely hidden from the review but only sorted into their own "unfinished" section. This design decision was made because access to unfinished stories within the sprint review can help discuss dependencies and interrelations between finished and unfinished stories.

In order to preserve feedback from the attendees, the meeting sidebar provides a note-taking feature. These notes may be anything worth mentioning for later use, for example, remarks concerning the product increment, more details on certain requirements, or arising ideas for new user stories.

When all finished stories have been presented, and the feedback has been gathered, the guided meeting tour can be switched to the *Backlog Refinement* step (see Figure 9.26). This will bring up the product backlog so the attendees can mutually work on the collected feedback and refine existing user stories or create new ones. Refinement and creating new stories, however, are supposed to be done quickly and in bullet points to keep the meeting short and engaging. After the review meeting, the product owner is responsible for the detailed revision of the backlog, including reprioritizing its items.

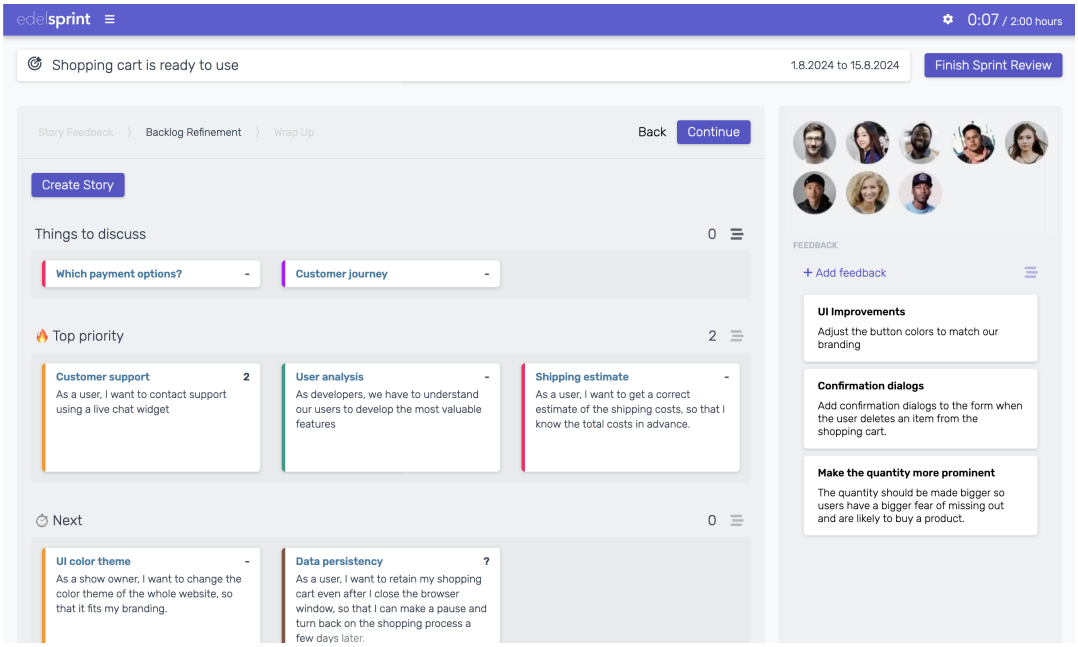


Figure 9.26: The "Backlog Refinement" step of the sprint review

In order to counter the problems of Section 6.2.5.8, edelsprint does not automatically transfer the unfinished items to a new sprint but transfers them back to the backlog. For this, it automatically creates a new section at the top of the backlog with the name of the completed sprint, in which the unfinished items can be revised according to the newly gained knowledge.

—

Demo video of the sprint review:
https://youtu.be/YOQOcC65zvA?si=qjtvCc92eFqZZ3_3

9.3.6 *Sprint Retrospective*

As the investigations in Chapter 7.2.2.9 show, the tool-side support for a sprint retrospective is severely limited in the selected agile ALM applications. It is essentially restricted to creating shared text documents in which the results of a retrospective can be recorded. Dedicated functions to support the execution of a retrospective, such as detecting problems in the sprint or identifying improvement potentials, are not offered. The data for analyzing the sprint and reflecting on teamwork is also completely missing. Unsurprisingly, there are also no functions for managing identified improvements, i.e., concrete actions for optimizing the development process.

In contrast, edelsprint provides several dedicated features to work against the identified sprint retrospective problems.

To address the problem of low sprint retrospective attendance (see Section 6.2.8.1), edelsprint has been designed to capture two metrics for any of the obligatory Scrum events. For any Scrum event, it tracks the *duration* and the *attendance* of individual team members. Based on this, it calculates an *attendance score* that is visualized on the overview page of each meeting type. This is intended to increase awareness about deviations from the Scrum rules by making meeting durations and attendance rates transparent to the whole team.

To address the issue of poorly facilitated (see Section 6.2.8.2) and energy-draining retrospectives (see Section 6.2.8.3), edelsprint furthermore provides guided retrospective formats in four steps, which include several *feedback mechanisms* to constantly gather insights about the sprint process and identify improvement potentials.

The first step is called *Set the Stage* and utilizes the common approach of beginning the retrospective with an icebreaker event to get everyone in the mood to reflect on the sprint process. For this, edelsprint provides a quick *satisfaction poll*, where attendees can ex-

press their satisfaction with the sprinting experience on a 5-point scale, represented by different emoticons (see Figure 9.27).

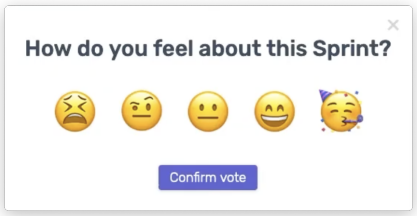


Figure 9.27: Satisfaction poll

The result of this voting, which is executed similarly to the story point estimation (see Page 272), is carried out via the attendees' mobile phones and is then displayed graphically (see Figure 9.28), thus making it possible to quickly form an initial impression of whether the participants' assessments are similar or whether there are possibly larger discrepancies in their perception of the sprint, which overall should help to set the stage for further analysis, as suggested by the Scrum expert Kenneth S. Rubin [226, p. 382].

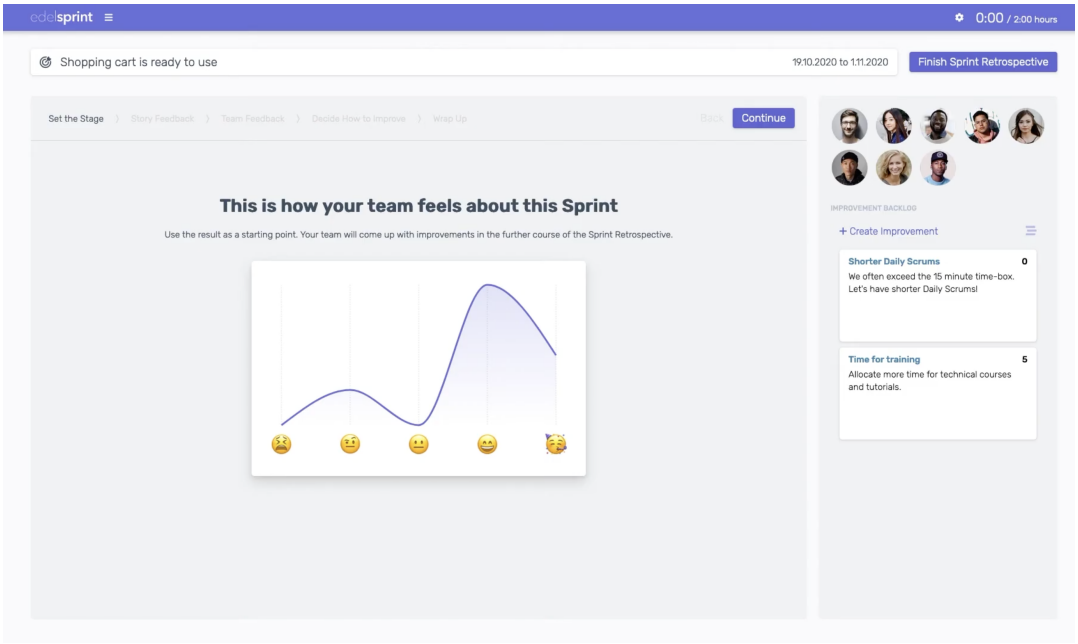


Figure 9.28: The "Set the Stage" step of a sprint retrospective

The second step is called *Story Feedback* and provides a dedicated view of the feedback given to finished and unfinished stories of a sprint, which is captured in the following way. When a story is part of a sprint, its details view contains a story feedback section below the definition of done (see Figure 9.29). Feedback to a story is independently saved for each team member and consists of an emoticon to express the feeling about the story and a text field to provide details.

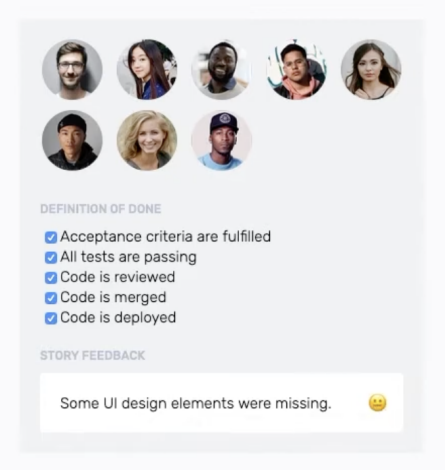


Figure 9.29: Feedback given to a user story

Since feedback is stored independently for each user, the story feedback section only shows the feedback provided by the currently logged-in user. As long as the sprint is running, this feedback can be changed. Only during the sprint retrospective, the collected story feedback will be accessible to other team members and shown to the whole group via the *Story Feedback* step (see Figure 9.30).

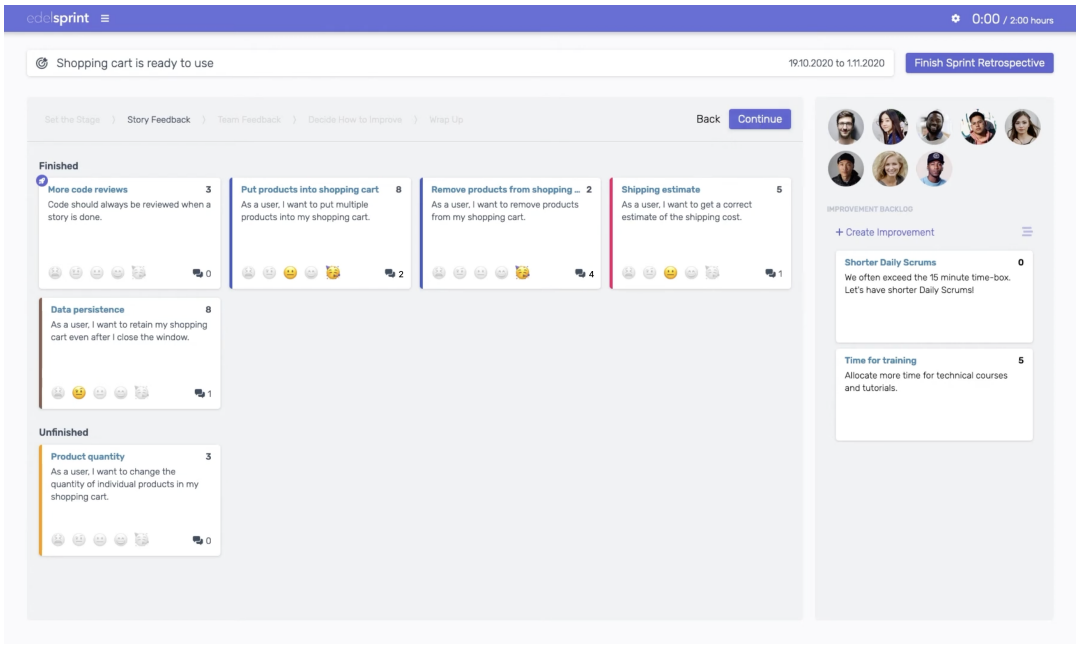


Figure 9.30: The "Story Feedback" step

Here, each story shows its user feedback responses through highlighting the corresponding emoticons at the bottom of the card next to the number of feedback responses. This way, the group can easily identify *different opinions*, which means that different emoticons are

highlighted, or *consent opionions*, which means that there are multiple responses but only one emoticon is highlighted (see Figure 9.31).

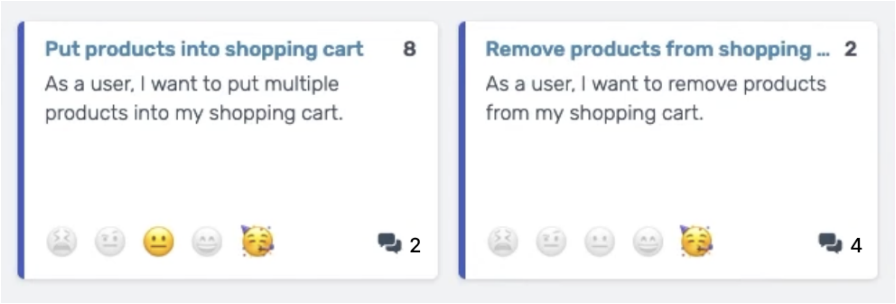


Figure 9.31: Different opinions (left) vs. consent opinions (right)

When clicking on the emoticons, the team can further investigate the feedback given to a story and see the provided comments. However, to increase willingness to incorporate user feedback into the sprint retrospective and minimize feedback bias due to participant uncertainty, results are presented anonymously, as shown in Figure 9.32.

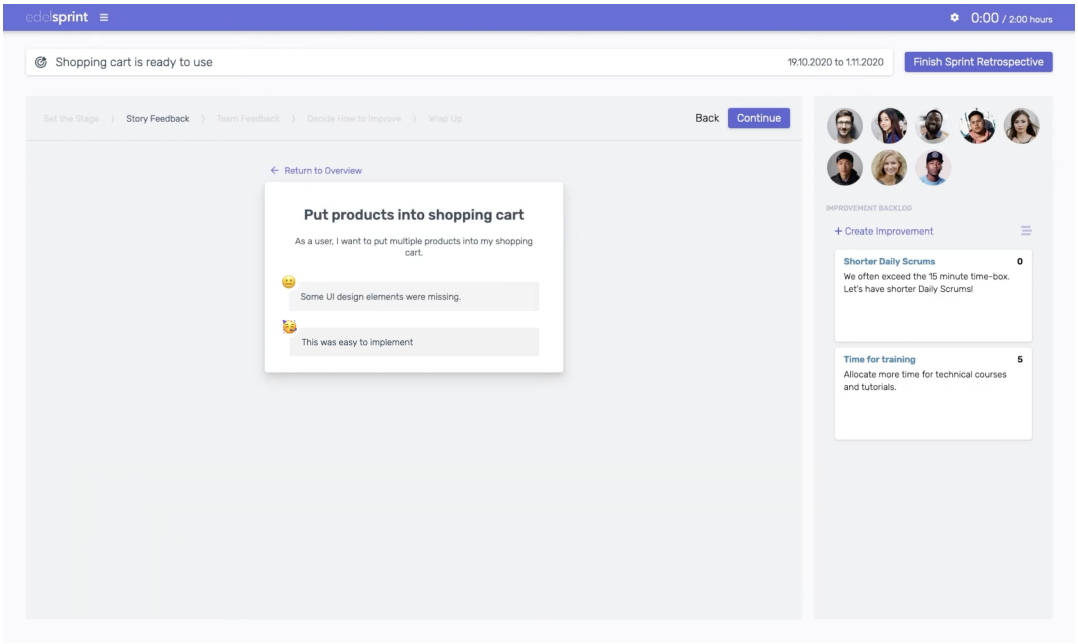


Figure 9.32: Anonymized feedback given to a story

Overall, the story feedback feature allows team members to capture observations and reflections on their work in real-time. Recording feedback directly on the related user story ensures that insights are not lost or forgotten during a sprint. Additionally, it should allow for more efficient retrospective analyses, as the Scrum team can quickly identify problematic stories and discuss the areas of potential improvements in the work.

While story feedback is on the level of user stories, the third step of the retrospective is called *Team Feedback* and provides the results of a higher-level and more general feedback mechanism. For this, edelsprint uses the *Start-Stop-Continue* method, which is a survey that the Scrum master can initiate during the sprint. The team members then receive an email with an access link to participate in the survey. Using this access link, a team member can provide feedback and share ideas about things that should be *started*, *stopped*, or *continued* to improve the sprint process. As long as the sprint runs, this feedback can be edited using the personal access link.

During the sprint retrospective, the anonymized results of all team members are shown in the *Team Feedback* step (see Figure 9.33). This allows teams to quickly gain insight into problems in the collaboration, which they can then jointly address at an early stage.

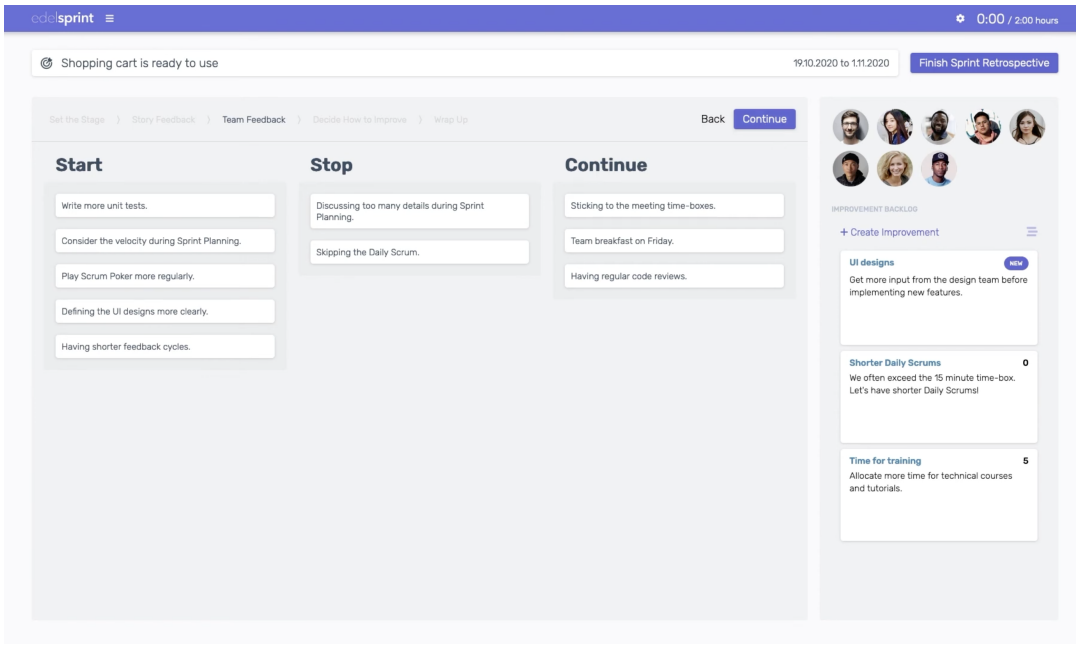


Figure 9.33: The "Team Feedback" step

During all stages of the retrospective, the team can use the provided feedback to create insights and decide on improvements, which can be recorded in the *improvement backlog* visualized in the meeting sidebar at the right (see Figure 9.33).

In order to prevent the typical Scrum problem of incorrect insight handling (see Section 6.2.8.4), the retrospective finally offers a *Decide How to Improve* step. This step aims to mutually work on the identified insights and transform them into executable improvements. While improvements are slightly differently visualized, they are nonetheless equally treated as user stories, i.e., they have acceptance criteria, tasks, notes, and can also be estimated since working on improvements is an effort that should be considered during sprint planning.

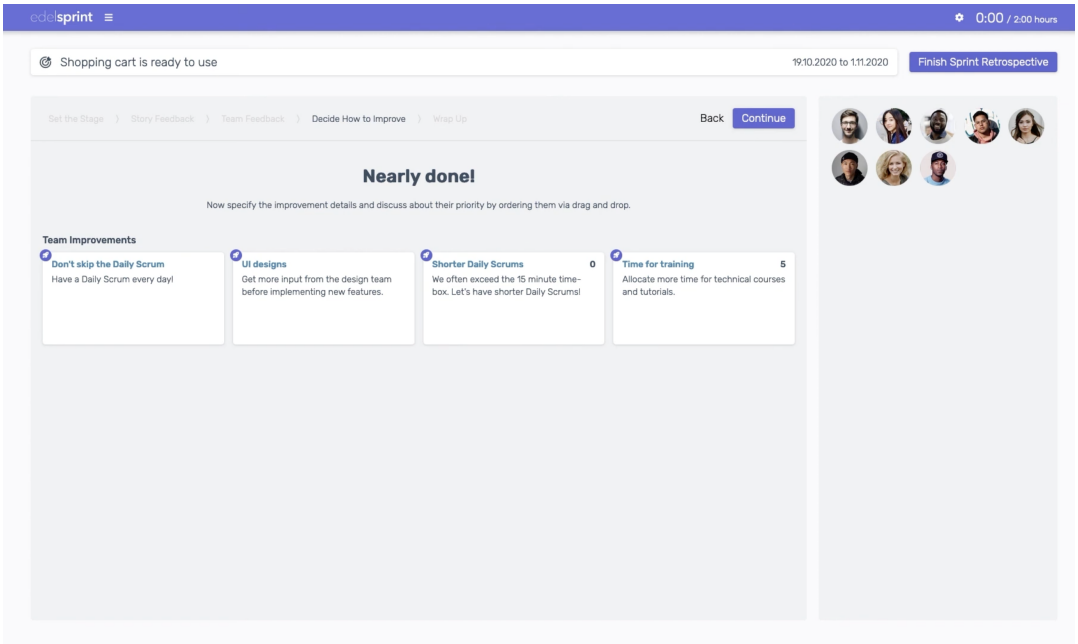


Figure 9.34: The "Decide How to Improve" step

For that reason, the improvement backlog is also shown as the first step of the sprint planning event (see Section 9.3.2), in which one or two improvements should be selected for the next sprint to create a sustainable teamwork optimization.

—

Demo video of the sprint retrospective:
https://youtu.be/LJTpZkLM-xQ?si=y1vCd336UTdRqmGY

9.4 EVALUATION AND CRITICAL DISCUSSION

Before concluding this thesis, this section will evaluate the implemented system and discuss how it can positively impact the previously identified Scrum challenges and issues, as well as the still-existing drawbacks and limitations.

9.4.1 *Iterative Evaluation during Implementation*

A unique feature of this thesis is that the evaluation of the proposed system was done to a large extent simultaneously with its development and not - as it is often the case - exclusively afterward.

This is due to the special circumstances that, as mentioned in Chapter 5, the prototypical development took place in the project group MAGICIAN, whereby the participants of the project group were both

developers *and* end users of the prototype. Since they worked according to the Scrum rules and used the prototype's current status for their own planning and execution of further development (see Figure 5.1 on Page 126), the evaluation thus took place continuously during the usual sprint work.

After the newly developed functions had been used in a sprint, they were subsequently discussed in the sprint review regarding their usability, and improvements were jointly developed, which were recorded in the backlog. Through this *Plan-Do-Check-Act* cycle, which is already given by the correct implementation of the Scrum framework, the prototype was successively improved, adapted, and expanded to the needs of a team working in accordance with the Scrum rules.

An excellent example of this iterative approach can be seen in the development of the planning poker feature, which is used in sprint planning to jointly estimate the workload of a user story (see Chapter 4.8.2.2). This feature was revised four times in the course of development, as explained in the following.

In the *first iteration*, planning poker was implemented on mobile devices so that the display showed a card over its entire surface. The user could use a swiping gesture to go to the next card and mark it as a selection with a tap. However, during the effort estimation in the subsequent sprint planning, this implementation turned out to be less user-friendly, as the smallest poker card was displayed on the mobile devices first with each new vote, so many swiping gestures had to be made to get to higher cards.

Therefore, the planning poker feature was adapted in the *second iteration* so that swiping gestures were dispensed entirely. Instead, all poker cards were shown simultaneously on the display of the mobile devices, whereby a card could again be selected by tapping. In the sprint review, however, half of the participants found this version difficult to use, as their displays were relatively small, and the fact that all cards were shown at once made it challenging to select them easily.

As a result, the *third iteration* combined the display of all planning poker cards at once, which was triggered when a mobile phone is held horizontally in landscape format, with the ability to swipe through individual cards, which was triggered when a phone is held vertically. While this approach satisfied the group members regarding the feature's interaction usability, the sprint review revealed that some members found it difficult to decide on a planning poker value when estimating a user story because of a missing estimation reference.

Hence, the *fourth iteration* introduced reference user stories. These are previously estimated user stories, randomly selected from the past three sprints, which show up on the user story's details page when

planning poker is triggered on the desktop interface. Since this interface is displayed to all attendees via a projector, the reference cards are shown to all group members as long as they take their vote using their mobile phones. This approach sufficiently simplified the estimation process, so no further iteration was needed for improvement.

With members of the project group being both developers and final users of the developed system, and due to the inspection and adaptation approach of Scrum, the system was continuously evaluated in a tight feedback loop. Moreover, this setting allowed the project group to experience Scrum in a very particular manner, which was honored with a teaching award from the Paderborn University[14].

However, it must be noted that the merging of developer and end-user roles also always carries the risk of a possible bias in the assessment of one's own development results. For example, someone trying hard to implement a feature successfully might prejudice its suitability when actually using it in the following sprint. As a result, the assessment during the final sprint review can be biased when the developer might argue for keeping a feature because of being attached to it since so much time was spent on developing it.

Therefore, special attention was paid to the students' working environment by establishing team-building measures, encouraging people to try new things, and living an open culture of error to welcome change. Moreover, to give more reserved people a voice, the sprint retrospective also included anonymous feedback polls to measure participants' satisfaction and reveal possible hidden needs or problems that might negatively influence evaluating the development.

Last but not least, the evaluation scheme of the project group was communicated with the participants in advance to make clear that the assessment does not consider contributions to what is considered the end result. In other words, an evaluation was neither based on lines of code nor on developed features that made it to the final prototype. Rather, the evaluation of a participant was based on how he or she contributed to the strengthening of the Scrum team and to the progress of the development, whereby progress can also mean that things that have already been developed have to be revised and, if necessary, even completely reworked.

14 https://www.uni-paderborn.de/universitaet/bildungsinnovationen-hochschuldidaktik/lehrpreis/lehrpreis-2015

9.4.2 *Heuristic Evaluation*

In addition to constantly evaluating the prototype during the project group's own Scrum process, it was also tested using two usability evaluation methods.

The first method used is *heuristic evaluation*, in which course a user interface is examined by usability experts, taking into account well-established usability principles, e.g., the "10 Usability Heuristics for User Interface Design" by Jakob Nielsen [191] or the "8 golden rules of interface design" by Jakob Shneiderman [252].

Heuristic evaluation was already introduced in Chapter 7, and as explained in Section 7.1.3, each expert examines the interface alone and independently of the others, which is intended to exclude mutual influence.

During the evaluation, which can be based on a typical application usage scenario, the experts examine the user interface's components, dialogs, and interaction elements and look for violations of the chosen usability principles. The outcome is a list of identified problems, including justifications for each problem explaining which principle is violated and their level of severity, i.e., the "expected impact on the users," taking into consideration the frequency and persistency of a problem [187].

To evaluate the prototye, the Nielsen heuristics were chosen (see Page 177), which are recommended to be used by three to five usability experts for having the greatest return on investment regarding the evaluation effort on the one hand and identifying as many different issues as possible on the other hand [193]. The investigation was conducted by four master students who previously successfully passed the course "Usability Engineering" at the University of Paderborn and hence were well familiar with the Nielsen heuristics and the heuristic evaluation method. One of those students further used heuristic evaluation to compare the developed prototype against the Scrum tools mentioned in Section 7.2.2 as part of his master thesis [177].

Before conducting the heuristic analysis, a fictional usage scenario covering most aspects of the Scrum framework was created, which has been explained on Page 174. The investigators were guided along this usage scenario and used the Nielsen heuristics to analyze the developed system in terms of the implemented features, from backlog grooming and sprint planning to the daily Scrum and sprinting towards the sprint review and sprint retrospective.

Overall, the evaluators identified the following seventeen problems. Six problems were classified as "cosmetic," i.e., having no impact on executing the task but may be nonetheless disturbing for the user,

while eleven problems were classified as "low," i.e., affecting task execution in a way that the user can still perform the task but may require more (cognitive) effort than necessary.

Cosmetic:

1. The button for adding tasks is styled differently from the rest.
2. The button for saving acceptance criteria can be clicked, although no input has been made.
3. The input field for adding a section name has no label.
4. Dragging a section is only possible via the drag handle, not the entire section header.
5. Cancelling a sprint requires navigating to another menu item.
6. There is no visual signifier in the sprint list for unfolding a sprint to see its items.

Low:

7. Creating a section in the backlog cannot be canceled directly.
8. Product backlog items cannot be filtered by category.
9. The category names do not show in the settings.
10. The backlog items only show colors but no category names.
11. A planning poker vote can be made without selecting a card.
12. The reference cards during planning poker can stem from different projects.
13. Creating a sprint without an end date results in a technical error message.
14. Tasks names are cut off on the board when they are too long.
15. Tags are cut off in the board.
16. The task list does not filter for unassigned tasks.
17. The task list does not filter for someone's own tasks.

The fact that only less severe usability problems were identified may be surprising at first. However, it must be considered that the investigation was conducted at the end of the project group when the students stopped working on it, i.e., the prototype was continuously evaluated and optimized during its development, while likewise, the participants tried to finish the implementation and deliver the prototype in as flawless a condition as possible[15].

15 All of the remaining problems identified via the heuristic evaluation were fixed in the funding program, which will be explained in the following section.

The second usability evaluation method used is *usability tests*, which were conducted during a funding program and will be explained in the following section.

9.4.3 *Usability Tests and Expert Interviews in Preparation for Market Entry*

Due to the prototype's overall positive results and the already extensive feature set, the idea was born to further develop the system into a marketable product. As a result, the funding program "START-UP-Hochschulausgründungen NRW" was chosen, which idea is "to further develop research results/know-how with great market potential and a convincing business concept and to convert it into the foundation of an own company."[16]

In the course of the funding program's approval process, the further development of the prototype was evaluated by a commission of experts from both economic and technical points of view and found to be positive in 2019. Four persons on this commission held leading positions in IT companies or consulting firms and were well familiar with agile methodologies. They confirmed the Scrum problems pointed out in Chapter 6 with their own experiences and saw great potential to solve some of them with the presented system, which is why the funding application was granted with a volume of 240.000 Euro provided by the European Union (EFRE)[17] and the state North Rhine-Westphalia.

The technical work packages of the funded project included a reimplementation to ensure the scalability of the software to many users with strict data separation, as well as an extension of the feature set with AI functions, which will be briefly addressed in the outlook of this thesis, but also further usability tests and evaluations with pilot customers to advance the market readiness of the product.

Since the management of the backlog and the processing of tasks is an essential part of agile ALM software, special attention was paid to these functions, and a usability test of the prototype was carried out with ten people at the beginning of the funding project, in which the test subjects were guided through a previously defined scenario.

This scenario included:

1. the *creation of a user story in the backlog*, including the definition of acceptance criteria and the definition of a category,

16 Translated from the funding program website [273]
17 Europäischer Fonds für regionale Entwicklung [144]

2. the *preparation of a sprint*, including the creation of a new section in the backlog and adding backlog items to this section,
3. the *creation of a new sprint*, including the definition of a sprint goal, sprint duration and a selection of backlog items from the previously created section, and
4. the *execution of typical sprint work* including creation of tasks, definition of responsibilities, and processing of tasks in the board.

Using the "Thinking-Aloud" evaluation method [127], it was possible to understand how the test subjects think while using the software, leading to further usability optimizations.

After finishing the re-implementation work, the product launch took place at the end of 2020 in form of a *minimum viable product*[18]. At the beginning of 2021, in the course of ongoing product validations, a user experience survey was conducted when the first 150 users were reached, and a *Net Promoter Score* (NPS) was determined on the basis of 68 collected data points.

While the NPS is not an official usability evaluation method, it is "a loyalty metric that correlates well with the perception of usability" with the benefit of being "easy to understand and administer" [84].

The evaluation is made by asking users to answer the question, "How likely are you to recommend this product to a friend or relative on a scale of 0 (do not recommend at all) to 10 (highly recommend)?"

All answers are later grouped into the following three categories:

- **Promoters** are at the top of the scale with responses of 9 or 10, thus indicating high satisfaction and a strong likelihood of a product recommendation.
- **Detractors** are at the lower end of the scale with responses between 0 and 6, thus indicating dissatisfaction and likely criticism.
- **Passives** are responses of 7 or 8 indicating moderate satisfaction but low likelihood of recommendation.

The NPS is then calculated by subtracting the percentage of detractors from the percentage of promoters:

$$NPS = \frac{Promoters \times 100}{TotalRespondents} - \frac{Detractors \times 100}{TotalRespondents}$$

18 The concept of a *minimum viable product* (MVP) comes from the *Lean Startup* methodology, emphasizing the importance of *learning* during new product development. According to Eric Ries, an MVP is the initial version of a new product that enables a team to gather the most validated learning about customers with the least amount of effort. This validated learning is in the form of whether customers will actually purchase the product or not.

As can be seen, *passives* do not contribute to the score except being part of the *total respondents*. That is because they are neither likely to promote the product to others nor likely to advise against using it, whereas *promoters* and *detractors* both have a direct impact on the NPS.

Interestingly, the cutoff points are strict, limiting promoters to a score of 9 or 10, whereas detractors have a much larger range from 0 to 6, which results in passives being beyond the mathematical midpoint of the scale. However, this is because raters "tend to be generous and give fairly high scores," so the expected level of satisfaction is indeed not 5, but the "perceived mid-point" on the 0-10 scale has been identified as 7 [84].

Theoretically, the NPS can range from -100 (only detractors) to 100 (only promoters). However, it is more restricted in practice because, usually, not all participants tend towards either of the extreme values. While considering positive values as "good" and negative values as "bad" may be obvious, interpreting the NPS is not as simple as it seems at first glance.

A loosely agreed-upon standard [215], which is also suggested by the creators of the NPS metric, *Bain & Company*, is the following classification:

- 0 - 20: "good"
- 21 - 50: "great"
- 51 - 80: "amazing"
- 80+: top percentile, dominant market leader

For using this classification, also known as *absolute NPS*, it is important to know that the score is compared to the results across all industries. However, the average NPS widely varies across different industries, so an NPS of 40 might outperform the market leader of one industry, whereas it might turn out to be the worst score in a different industry. Hence, it is usually more meaningful to benchmark the NPS with competitors of the same industry, which is known as *relative NPS*.

For the evaluation mentioned above, edelsprint received an NPS of 30, which, as an absolute score, can be considered a "great" result.

Considering that the NPS of software products has been identified to range from -26% to 40%, with an average of 15% [215], the overall interpretation of the received NPS must be considered positive, representing strong user loyalty and high perceived usability.

9.4.4 *Discussion*

This section will critically examine the proposed system for addressing the Scrum challenges and problems identified in Chapter 6 and solving the existing limitations of the investigated agile ALM tools, as described in Chapter 7.2.2.

Afterward, the limitations of the developed system will be discussed before closing this thesis in Chapter 10 with an outlook showing possible improvement potentials and expansion options.

9.4.4.1 *Addressed Scrum Issues and Challenges*

A central outcome of the Scrum challenges examination of Chapter 6 is that many severe issues are related to the obligatory Scrum meetings, which also were identified in Chapter 7.2.2 as being hardly supported by existing agile ALM tools.

Regarding the sprint planning meeting, the developed system addresses most of the identified issues, as shown in Table 9.2. This is made possible primarily by new features that are absent in the investigated agile ALM tools.

	ISSUES OF SPRINT PLANNING
✅	Overestimating the sprint scope
✅	No slack time
✅	Too detailed planning
✅	Not using a sprint goal
✅	Ignoring the definition of ready
✅	Unfinished stories spill over to the new sprint
✅	Not considering improvements
✅	No team-based decisions
✅	Ignoring technical debt
❌	Dominant product owner

Table 9.2: Addressed sprint planning issues

Overestimating the sprint scope and **no slack time** are made transparent through a warning message that is shown to the whole team during the sprint planning event when the sum of story points selected for the sprint exceeds a certain threshold. This threshold is re-calculated for each sprint planning and represents the average of story points finished within the latest three sprints plus a 10% slack time buffer, as explained in Section 6.2.5.4.

In order to maximize efficiency and prevent **too detailed planning**, the developed system shows a meeting timer visualizing the time past and the time available according to the rules of the Scrum Guide. Together with the sum of the stories already selected for the sprint, the team can thus estimate at any time whether they will manage with the remaining time of the meeting or shorten discussions about unnecessary details if necessary. In addition, the backlog items have been designed according to the story card metaphor with deliberately reduced space for text input to avoid written over-specification of requirements and instead foster the use of acceptance criteria as a lightweight specification approach.

The issue of **not using a sprint goal** has been faced by dispensing a technical designation of a sprint, e.g., by using a numbered identifier (e.g., "Sprint 1," "Sprint 2," etc.) and instead making the sprint goal freely definable and a central element by visualizing it prominently in the user interface of both the sprint planning event and later in the board when the sprint has started.

Since **ignoring the definition of ready** was identified as a common problem of the sprint planning event, the developed system offers a freely configurable definition of ready, displayed as a separate checklist for each backlog item. This allows each backlog item to be clearly identified as to whether it has already been sufficiently specified, prepared, and understood by the team to be included in a sprint.

The common issue that **unfinished stories spill over to the next sprint** is prevented by the developed system by automatically making all unfinished stories of a sprint return to the backlog in a new section labeled as "Unfinished items from sprint <Sprint Goal>" when a sprint is closed. While all elements of an item are preserved, i.e., the title, description, notes, list of acceptance criteria and tasks, the definition of ready is reset, which shall further ensure that the element goes through the same steps of planning before it is included in a sprint again at a later time.

Not considering improvements during a sprint has been identified as a problem that renders the efforts of the retrospective for identifying insights obsolete and limits the team's chance for working and spending time on self-improvement. The developed system emphasizes the importance of including an insight into the sprint by integrating the selection of improvements into the sprint planning process. Even before the actual backlog items are discussed, the improvement backlog is first displayed, and selecting at least one item from it is encouraged.

Features targeting the issue of **no team-based decisions** first of all include the backlog item's list of acceptance criteria, which shall ensure that the criteria of acceptance are known to the whole team before the item is considered for a sprint. While the acceptance criteria

are the pure responsibility of the product owner, this does however not mean that the development team has no decision-making power. This is because it takes the whole team to agree to check off items in the definition of ready, which usually includes accepting the acceptance criteria. The development team, therefore, has a kind of veto right through the definition of ready and can declare the acceptance criteria as not (yet) ready, which blocks the inclusion of the item in the sprint. In addition, the planning poker feature is available to the development team to estimate the implementation effort of a story. Overall, this encourages team-based decision-making in the sprint planning because while the product owner remains ownership of what has to be achieved in the sprint, it is up to the development team to estimate the reasonable scope and reject stories that would put too much stress on the team, for instance, due to unclear acceptance criteria.

The issue of **ignoring technical debt** is addressed by the developed system in two ways. At first, the backlog can be configured and used so that the team takes care of technical debt during sprint planning. This could be done by defining a "technical debt" story category and collecting all stories belonging to that category in a dedicated section on top of the backlog. Therefore, this section will be presented during the sprint planning event, encouraging the team to incorporate refactoring tasks and cleaning the code base into the usual sprint work. Second, and most importantly, sprint backlog items can be individually checked against the definition of done, which is prominently displayed on each story details page and contributes to ensuring that cleaning the code is part of the implementation work.

In contrast to the previously mentioned, the issue of a **dominant product owner** is not particularly addressed by the developed system. That is because this problem stems from social and cultural mismanagement within a company as well as a false understanding of agile values, as described in Section 6.2.5.2. While the developed software fosters collaboration according to the rules of Scrum, it cannot prevent the misbehavior of the product owner or any other team member, strictly violating the framework's rules. For that reason, the presence of a Scrum master, acting as an agile authority and occupying the role of a servant leader, as described in Section 4.4.4, must be ensured.

Regarding the daily Scrum, five of the seven issues are particularly addressed by the developed system, as shown in Table 9.3.

In order to prevent teams from having **no routine** for the daily Scrum meeting, the developed system provides a guided step-by-step process through any of the Scrum meetings. For the daily Scrum, this includes extended support for making each team member answer the three questions "What have I done yesterday?", "What will I do today?" and "What are my impediments?" using the features of the implemented board (see Chapter 9.3.3).

	ISSUES OF THE DAILY SCRUM
✅	No routine
✅	Low team spirit
✅	People don't listen to each other
✅	Beginning discussions and exceeding the timebox
✅	Low willingness to help
❌	The daily Scrum downgrades into a status report meeting
❌	Command and control by the management

Table 9.3: Addressed daily Scrum issues

While this clear structure of the process shall foster building a routine, it shall also lay the foundation for targeting the issue of **low team spirit** and the common problem that **people don't listen to each other**. Both problems should benefit from the clear flow of the given structure of the daily Scrum. They are furthermore addressed by the dedicated meeting UI that is specifically designed to be shown on projected screens, allowing participants to easily read information from a further distance, thus enhancing the prerequisites for actively participating in the event.

Reducing visual elements and only showing relevant information during the daily Scrum together with the meeting counter visualizing the past and remaining time shall further prevent the problem of **beginning discussions and exceeding the timebox**. Keeping the daily Scrum short should also contribute to increasing the attendees' attention and counter **low willingness to help** others. Moreover, the developed system introduces "blocker" states, which visualize tasks that are blocked for whatever reason and cannot be further processed by the assigned person. The developed system ensures that all of these blocker tasks are shown in the daily Scrum, thus highlighting the need for help. In contrast to other tools, tasks can also be assigned to multiple persons. With this, people willing to help can be easily added to tasks on the fly while keeping the flow of the meeting going.

While the daily Scrum of the developed system has been first and foremost designed to fulfill the needs of the development team, it does not particularly address the issue that **the daily Scrum downgrades into a status report meeting** or the problem of **command and control by the management**. Both stem from mismanagement and a false understanding of the agile principles and the roles of Scrum, according to which the development team is supposed to be self-organized and solely responsible for turning the items of the sprint backlog into a working increment. Similar to the issue of a dominant product owner of the sprint planning, these problems should be, therefore, addressed and handled by a professional Scrum master.

Regarding the sprint review, three of the four identified issues are addressed by the developed system, as shown in Table 9.4.

	ISSUES OF THE SPRINT REVIEW
✅	No discussions, no feedback
✅	Development team is not focusing on relevant content
✅	Treating the review as venue of approval
❌	Omitting the demo for technical implementations

Table 9.4: Addressed sprint review issues

To counter the problem of **no discussions and no feedback**, the user interface of the sprint review meeting is (as all other meetings) designed with elements and font sizes that can be seen from a greater distance to ensure that any attendee can read properly and concentrate on what is presented. In addition, the meeting allows users to quickly collect feedback from the attendees, which is omnipresent and can be worked into the backlog later on or on the fly, even during the meeting.

The issues of the **development team not focusing on relevant content** and **treating the review as the venue of approval** are both targeted by clearly differentiating between finished and unfinished sprint backlog items in the sprint review. An item is automatically classified as finished when it is part of the "Done" column of the board, and all points of its definition of done are checked. In any other case, an item is considered to be unfinished. That way, the development team can focus on presenting what has been finished while stakeholders can focus on giving feedback to derive new ideas for improvement. The clear distinction between finished and unfinished items shall further encourage the product owner to establish an ongoing review process throughout the whole sprint instead of treating the meeting itself as the venue of approval. Otherwise, the sprint review meeting would show all items as unfinished, which will certainly lead to questions from the stakeholders present.

However, since the developed system only focuses on the aspects of project management and the unique components of Scrum, it offers no integrations for other tools. It hence cannot track whether the team is **omitting the demo for technical implementations** or demonstrating results in general. As a result, the Scrum master must still ensure that the review is executed correctly.

Regarding the sprint retrospective, the developed system addresses all of the identified issues, as shown in Table 9.5.

ISSUES OF THE SPRINT RETROSPECTIVE	
✅	No retrospective or low attendance
✅	The retrospective is poorly facilitated by the Scrum master
✅	Depressing and energy-draining retrospectives
✅	Incorrect handling of identified insights

Table 9.5: Addressed sprint retrospective issues

What compensates that teams have **no retrospective or low attendance** is that the developed system clearly shows the supposed sprint process, including all of the obligatory meetings. While this does not prevent intentionally skipping the retrospective, it brings it into the same focus as all other events, thus stressing its importance. In addition, the attendance of each team members is tracked for each meeting type and made transparent to the whole team by visualizing it on the event page, which should motivate to keep the attendance rate high.

The issues that **the retrospective is poorly facilitated by the Scrum master** and may become **depressing and energy-draining** are addressed by several features which are intended to collect feedback from the team to derive valuable insights for improvement, yet keeping the meeting short and efficient. Each retrospective starts with an icebreaker poll, which serves the purpose of getting a quick snapshot of how the team feels about the sprint. Afterward, the team can use the story feedback feature to derive insights from working on the stories of the sprint, as well as the team feedback feature utilizing the Start-Stop-Continue method to manage ideas for improving as a team. All of this collected data is intended to prevent **incorrect handling of identified insights** and to support deriving improvements. By providing an improvement backlog and making it part of the next sprint planning event, the system furthermore puts particular emphasis on improving as a team according to the Scrum rules.

Regarding the product owner role, the developed system addresses four of the six identified issues, as shown in Table 9.6.

ISSUES OF THE PRODUCT OWNER ROLE	
✅	Bad preparation of stories
✅	No testable acceptance criteria
✅	Disregard of grooming meetings
✅	Subsequent modifications of the sprint backlog
⭕	Lack of authority and no ordering of the backlog
❌	Absent product owner

Table 9.6: Addressed issues of the product owner role

The provided features for implementing a custom definition of ready into the backlog management process target the issue of a **bad preparation of stories** together with other backlog management features, such as user story categories or draggable backlog sections for easily ordering stories according to their priority.

As described in Section 9.3.4, backlog sections furthermore play an important role in addressing the identified issue of **disregarding the grooming meeting** because they are used to highlight the results of backlog grooming sessions. As the developed system offers dedicated functionality for collaborative backlog grooming events, the stories discussed are automatically transferred to new sections and tagged with actions to simplify later refinement by the product owner.

In contrast to the investigated agile ALM applications, the developed system also addresses the common issue of **no testable acceptance criteria** by providing a dedicated list of acceptance criteria for each user story. By design, this list encourages the product owner to keep each criterion's description short and testable.

The issue of **subsequent modifications of the sprint backlog** is targeted not by preventing modifications at all because it cannot be ruled out that there may be good reasons to change the sprint backlog even though the sprint has already started, but by making subsequent modifications a very conscious decision by the whole Scrum team. That is because, in contrast to the investigated agile ALM tools, the developed system introduces dedicated collaborative meeting events, and modifying the sprint backlog is only possible during a sprint planning event, which is clearly visualized as the starting point of a sprint. So, subsequently modifying the sprint backlog requires the product owner to start another sprint planning event and move through all of its steps, as described in Section 9.3.2, thus making these modifications a conscious decision taken by the whole team instead of the product owner alone.

The issue of a product owner with **lack of authority and no ordering of the backlog** is partly addressed by introducing novel ordering and filtering features for the product backlog, i.e., draggable backlog sections and user story categories. However, the underlying problem of the identified issue is not the ordering per se, but the fact that it results from a product owner without authority not owning the product, hence not fulfilling the duties of the role, which might be for various mismanagement reasons, such as promoting former business analysts to become product owners, interfering decisions taken by higher management, or fragmentation of responsibilities, as described in Chapter 6.2.4.2. Altogether, these problems are related to the internal company organization and are therefore out of the scope of this thesis.

For the same reasons, the issue of an **absent product owner** is not targeted by the developed system, which is also rooted in incorrect appointments to the product owner role and a false awareness of the importance of this position for the success of the entire Scrum team.

In terms of knowledge management, both of the identified issues are addressed by the developed system, as shown in Table 9.7.

	ISSUES OF KNOWLEDGE MANAGEMENT
✅	No documentation of decisions
✅	Knowledge hotspots

Table 9.7: Addressed knowledge management issues

Having **no documentation of decisions** is targeted by providing a history of previous sprints, including the individual goals, the user stories contained, as well as their acceptance criteria, tasks, and notes. Furthermore, by using proper code committing standards, e.g., *pull requests*[19], developers can draw a connection between their work in the codebase and the user stories of a sprint, enabling other developers to track decisions about the course of development both at the user story level and the software code level.

Knowledge hotspots are addressed by making the work of everybody transparent to the whole Scrum team, and that is true for both the level of user stories and the level of tasks. As a result, it will become evident if a person only works on specific user story types, e.g., stories belonging to the "frontend" category, or is solely responsible for a specific task, e.g., "writing tests" for different stories. Both scenarios would be against the idea of being a cross-functional team, which is about sharing skills and having mutual knowledge, as explained in Chapter 4.4.3.

In terms of understanding Scrum, the developed system addresses one of the two identified issues, as shown in Table 9.8.

	ISSUES OF UNDERSTANDING SCRUM
✅	Scrum as a framework provokes ScrumButs
❌	Certification as the single source of knowledge

Table 9.8: Addressed issues of understanding Scrum

As described in Chapter 6.2.9.2, deriving from the Scrum framework often leads to harmful modifications, which in the end lead to negative results with regard to the agile work setting and overall team efficiency.

19 Pull requests allow developers to inform others about code changes pushed to a branch in a shared repository.

The developed system addresses this issue that **Scrum as a framework provokes ScrumButs** by offering functionality that is very much tailored to the official rules of the Scrum Guide. Beginning with supporting each of the obligatory Scrum meetings, followed by offering the necessary features for collaborative planning and sprint work, the developed system specifies the sequence of an ideal Scrum process. However, the developed system does not restrict users from having any deviations at all. For example, a team may decide not to conduct a sprint retrospective. While this "ScrumBut" is not actively prevented, the developed system will nonetheless show the deviation from the ideal process, thus making it transparent to the team.

The issue of treating a single Scrum **certification as the single source of knowledge** is not particularly addressed by the developed system. However, due to its design and guidance through the sprint cycle, it can serve as a helpful tool to learn about Scrum and its individual components. But still, an experienced Scrum master is absolutely necessary to overcome many of the issues presented in Chapter 6 and remove impediments from the Scrum team.

This is especially true for issues resulting from teams being embedded in waterfall-ish environments. Regarding this challenge, the developed system addresses three of the seven identified issues, as shown in Table 9.9.

	ISSUES OF WATERFALL-ISH ENVIRONMENTS
✅	Developers are left out from the requirement analysis
✅	Documents are still driving the development process
✅	Monitoring hours for reporting
❌	Business analysts become product owners
❌	Resource management and project culture
❌	Scrum team is not allowed to be cross-functional
❌	Customers are not participating

Table 9.9: Addressed issues of waterfall-ish environments

In order to counter that **developers are left out from the requirement analysis**, the developed system offers collaborative backlog grooming events, which take place at the tabletop and can be quickly started to foster discussions about the backlog between the product owner and members of the development team. In addition, developers are free to suggest new ideas to the backlog, which, when created, are displayed in a separate backlog section. This way, ownership of the backlog and its prioritization and ordering still belong to the product owner, while the whole team is invited to come up with their own ideas for product improvements.

To address the issue that **documents are still driving the development process**, the developed system provides an extensive set of features ranging from acceptance criteria lists and attachable images or other file types over backlog sections and user story categories to readiness checklists represented by the definition of ready, to ensure that the backlog can be operated as "single source of truth" in terms of managing product requirements.

As described in Chapter 6.2.2.7, another common problem for teams in waterfall-ish environments is **monitoring hours for reporting**, which results from project managers using working hours as calculation for features or project prices (see Page 144) and contributes to false expectations at the management level, because of reports and forecasts of features to particular dates. Since this creates the illusion of being able to know everything in advance and neglects the ever-existing amount of uncertainty, the developed system only provides story points as an estimation technique and, through this conscious design decision, prevents planning or reports from being based on a precise hourly calculation.

The issue that **business analysts become product owners** is not particularly addressed by the developed system because personnel decisions are out of the scope of a project management tool. However, it does address some of the resulting problems, e.g., the lack of responsibility to really *own* the product and drive the business by own decisions through extensive role-specific backlog management features, e.g., by only allowing the product owner to define and modify acceptance criteria, thus strengthening responsibility for product development decisions.

Likewise, this issues of bad **resource management and project culture** and when a **Scrum team is not allowed to be cross-functional** are not addressed and out of the scope of the developed system because these issues result from grown management structures within a company, limiting the potential for true agility, as described in Section 6.2.2.2.

The problem when **customers are not participating** in the feedback-response cycle is also, and for the same reason, not particularly addressed. While the developed system does provide dedicated features to conduct beneficial sprint review meetings for attending customers, it seems out of scope for software to affect the motivation and willingness of individuals to participate in a meeting in the first place.

9.4.4.2 *Limitations of the Developed System*

While the previous section summarized how the developed prototype addresses the identified Scrum challenges and issues from Chapter 6, this section will now discuss the remaining drawbacks and limitations.

As mentioned in the previous section, the developed system does not resolve all identified issues. Specifically, it does not address the problems outlined in Table 9.10.

	ISSUES NOT ADDRESSED
✗	Dominant Product Owner
✗	The daily Scrum downgrades into a status report meeting
✗	Command and control by the management
✗	Omitting the demo for technical implementations
○	Lack of authority and no ordering of the backlog
✗	Absent Product Owner
✗	Certification as the single source of knowledge
✗	Business Analysts become Product Owners
✗	Resource Management and Project Culture
✗	Scrum Team is not allowed to be cross-functional
✗	Customers are not participating

Table 9.10: Issues not addressed

Most of these problems result from former mismanagement and grown company structures, e.g., waterfall-ish environments. As described in Chapter 6.2.2, in many cases, these environments leave no room for agility and counteract the special demands of self-organizing, cross-functional Scrum teams. Hence, it must be said that the effect of a novel software tool, although tailor-made for Scrum, is nonetheless somewhat limited when the foundation for Scrum is not in place.

Regarding the addressed issues, it must be said that "addressed" does by no means imply that the developed system is a guaranteed *solution* to the identified problems. Rather, it means that it provides *dedicated features* for these problems, which were developed with regard to the Scrum-specific challenges of Chapter 6 and taking into account the analysis of existing agile ALM tools of Chapter 7.

The developed system, therefore, does not enforce correct work according to the Scrum rules but merely offers support in order to be able to carry out a development process close to the Scrum ideal, and with the help of mandatory meetings, roles, and artifacts. Although a violation of the rules is recognized at various points in the software and indicated by warnings, users can make a conscious decision not

to follow some Scrum rules correctly. For example, the sprint planning issues "overestimating the sprint scope" and "not using a sprint goal" cannot be entirely prevented by the developed software. Although a warning is displayed if the sum of the story points of a planned sprint exceeds a certain threshold, the sprint could still be started. Similarly, although a sprint cannot be created without a sprint goal, there is no check as to whether this sprint goal makes sense (or just consists of an arbitrary sequence of characters).

Compliance with the rules and ensuring that the development process runs as smoothly as possible should, therefore, remain the responsibility of the Scrum master, who must also continue to be responsible for protecting the development team from tasks that do not achieve their objectives, e.g., in the daily Scrum through command and control by the management (see Section 6.2.6.7) or preventing the team from being influenced by an overly dominant product owner (see Section 6.2.5.2).

This should be emphasized in particular because, during the market launch of edelsprint, it became apparent that some customers, after an internal company evaluation phase of the software, no longer intended to fill the Scrum master position due to the software-side support of the Scrum rules and thus wanted to save costs. However, this is strongly discouraged, and the developed system is by no means intended nor suitable to replace the work of a Scrum master, as this goes far beyond mere adherence to the rules, such as resolving impediments during a sprint or mediating between the development team and the product owner, which require interpersonal interactions and cannot be carried out by functions of a software tool, no matter how sophisticated they may be.

A further possible limitation of edelsprint is that the structure of the software is based on an ideal, i.e., textbook-accurate, Scrum process. This is reflected on the one hand in support of all relevant artifacts (user stories, product, sprint, and improvement backlog, the definition of ready, the definition of done, etc.) and on the other hand in the provision of functions to conduct all Scrum meetings interactively and collaboratively - from sprint planning to the retrospective.

As described in Chapter 9.3.2, each meeting has a clearly defined process specified by the software. For less experienced teams and newcomers to the field of agile development, this fixed structure certainly offers the advantage of quickly getting to know the Scrum framework and, at the same time, receiving guidance through the process and its mandatory rules. On the other hand, experienced teams may find the meeting process too rigid and want more flexibility about the predefined processes, making it challenging to meet the requirements of both sides.

For example, as described in Section 9.3.6, the sprint retrospective in edelsprint always follows the same step-by-step pattern. Starting with a short atmosphere survey as an icebreaker and introduction to the meeting, the story feedback, i.e., feedback from team members on individual user stories of the sprint, is then discussed, followed by the team feedback step, in which the Start-Stop-Continue method is applied to then fill the improvement backlog with new ideas and prioritize them accordingly for the next sprint.

As described in Chapter 4.7.4, this approach is based on the recommendations of experts but can also seem forced for experienced teams and become boring in the long run. Deviations from the "norm" can, as described in Chapter 6.2.9.2, quickly lead to a softening of the Scrum rules, so-called *ScrumButs*, which in turn can have negative consequences. Nevertheless, Scrum should be seen as a framework within which free modifications should be possible, such as a free and thus varied design of the implementation of a retrospective, to maintain the team's motivation to carry it out in the long term.

It is, therefore, once again the task of the Scrum master to design the retrospective in a varied and meaningful way. For experienced teams, this should also be possible without any problems despite the rigid process in edelsprint, as the content of the individual phases of a retrospective "Set the Stage," "Gather data," "Generate insights," "Decide what to do," which edelsprint is based on, can be skipped if necessary and replaced or supplemented by other methods. Numerous examples of different methods are collected by various experts and can be found at the website `https://retromat.org`, along with explanations of how teams can use them within the retrospective, either analogously with pen and paper or with the help of online whiteboards, and use them to self-reflect on their work.

A similar problem concerning the diversity of the target groups arises for the scope of the features developed. The developed system is clearly limited to the core functions of Scrum. Although the elements of the framework are mapped in their entirety, functions that go beyond this scope were deliberately omitted. This is because, as shown in Chapter 7.2.2.2, a large number of functions can lead to an overload of the user interface, which can have a negative impact on the simplicity of operation, which in turn can be detrimental to agile working.

However, depending on the team or project management requirements, there may be a need for more far-reaching functions. A typical example would be functions representing a higher planning level, such as *roadmaps* or *calendar functions* for planning project-related releases, which typically extend over several sprints. Planning at a strategic level is also widespread, e.g., using the *Objectives Key Result* (OKR) method, which is widely used in companies and serves to track the

achievement of strategic goals (objectives) through measurable performance indicators (key results). Such planning levels can only be represented to a limited extent with edelsprint, at best via the possibility of dividing the product backlog into different sections, whereby a section could, for example, contain all stories for a specific release.

Nevertheless, it must be clearly stated that the functions of edelsprint only focus on mapping the Scrum framework as accurately as possible and are also limited to this. For the reasons mentioned above, a comprehensive extension that covers other aspects outside of Scrum has been avoided in order to keep the application deliberately simple. However, it is conceivable that data could be retrieved from edelsprint via a REST-API in the future, which would allow the sprint goal, user stories, and tasks to be integrated with external software products, such as *Jira*[20], which are better suited for higher planning levels of a project.

A direct connection to versioning solutions and code management systems, such as *Github*[21], is also not currently available. This would be useful to automatically link the planning level of a sprint, i.e., the progress of individual user stories, with specific changes to the source code, and thus be able to correlate them permanently. However, these relationships can also be achieved without a direct connection by adhering to the conventions of the development team regarding handling the versioning solution. For example, it is advisable to create a code branch for each user story of the current sprint that bears the name of the user story. The ID of the user story can then be placed in front of each published code change as a prefix to the commit message, creating a clear assignment between changes to the source code and the respective user story of the sprint.

With regard to the implemented multi-touch technologies, it can be said that they performed well in the evaluations and also received a lot of positive feedback in the expert interviews, but the lack of dissemination of these technologies still poses a challenge in terms of marketing the developed system.

Smartphones, which are used for user-specific interactions in collaborative scenarios (e.g., in the daily Scrum, as explained in Section 9.3.3, or as part of sprint planning, as explained in Section 9.3.2), are ubiquitous nowadays, and larger vertical displays are also increasingly being used in companies' meeting rooms to mutually discuss digital content. On the other hand, interactive tabletops such as those presented in Chapter 8 are hardly widespread and can only be found in very few companies. One of the main reasons for this is that tabletop application development differs significantly from conventional

20 https://www.atlassian.com/software/jira
21 https://github.com/

desktop software development. While the latter is characterized by classic WIMP-UIs and single-user functionalities, tabletop applications can represent multi-user scenarios, which poses unique challenges concerning the presentation of the user interface with people around a tabletop having different viewing angles and the processing of touch inputs from different people.

Despite an overall positive evaluation of the tabletop interface as part of the prototype developed, the commercial development of the system has therefore focused primarily on desktop, tablet, mobile, and wall interfaces. The support of a tabletop interface as described in Chapter 9.1 did not prove to be reasonable within the framework of the funding program, which focused on the commercial ability and further development of the system into a market-ready product, as the low prevalence of tabletops contradicted their overall higher development requirements.

10

SUMMARY AND CLOSING

This doctoral thesis examined the challenges and problems in implementing the Scrum framework, analyzed existing tools that support this methodology, and, based on these findings, showcased a new project management software utilizing Natural User Interfaces to better map the Scrum process.

With Part I aiming at setting the theoretical foundation for this thesis, Chapter 2 commenced with a historical overview of the evolution of computers and software development as an engineering discipline and derived some important lessons learned for today's development principles, e.g., that programming is a thoughtful and creative process and that it benefits from cross-cultural awareness when integrating diverse perspectives and talents into the software engineering discipline. While all history is characterized by outstanding people, the illustrated historical milestones also underscore the fact that collective efforts often outstrip the capabilities of individual experts, no matter how brilliant they may be. And so is the history of computation and software development, a history of the evolution from isolation to collaboration, which is now at the very core of agile methodologies like Scrum, where communication, teamwork, and sharing ideas are prioritized over individual prowess. Likewise, historical examples showed the incredible result of careful and correct preparation of tasks prior to the actual work execution, which became one of the most important software development principles because failures in the specification of requirements can have a tremendous impact in later coding phases and often lead to failing projects.

For this reason, Chapter 3 illustrated the evolution of software development lifecycle models, all of which address task preparation through different planning and execution strategies. While plan-driven development models focus on extensive upfront planning with all details contained in heavy documentation, the principles of iterative and incremental software development began to initiate a paradigm shift, as it turned out that plans, no matter how detailed and thought out, can always be thwarted by unforeseen events, which are difficult to handle because of the models' sequential nature, where requirements are more or less cast in stone from the beginning and difficult to change in later stages. On the other hand, lightweight approaches, which later should be coined "agile," can better compensate for unforeseen events

because of their adaptability to change resulting from short iterative and incremental development cycles.

Based on this and completing the theoretical foundations of this thesis, Chapter 4 provided a comprehensive examination of the Scrum framework as the most widely used agile development method. Covering its entire ruleset, including the different roles, artifacts, and sprint cycle events, it shows that Scrum is very logically structured and follows a straightforward empirical process model based on inspection, adaptation, and transparency.

Therefore, it is all the more surprising that Scrum, while easy to understand, is known for being difficult to master.

Getting to the bottom of this apparent contradiction was the objective of Part II of this dissertation. With Chapter 5 explaining the various research methods used for this thesis, Chapter 6 was guided by the first research question:

"What are typical challenges and issues for Scrum teams?"

By employing diverse research methods, including literature reviews, ethnographic studies, and interviews utilizing open questions and the laddering technique (see Section 6.1.3.2), this thesis provided new insights and contributed to a better understanding of the problems of agile Scrum teams. It identified a wide range of 42 Scrum issues, substantially more than previous research in this field, and clustered them into eight challenge areas:

- waterfall-ish environments (7 issues),
- the product owner role (6 issues),
- sprint planning (10 issues),
- daily Scrum (7 issues),
- sprint review (4 issues),
- sprint retrospective (4 issues),
- knowledge management (2 issues),
- understanding Scrum (2 issues).

However, the key takeaway is not only that the challenges in implementing Scrum are manifold and are particularly evident in waterfall-ish environments, the product owner role, and the collaborative Scrum events. But also that many of the issues identified are connected and show dependencies, which leaves the impression that the Scrum framework is built like clockwork, with many individual cogwheels that need to mesh together to make it work smoothly, as shown by Figure 6.2 on Page 139. If one fails, this can significantly impact other parts of the framework due to dependency effects, which can quickly lead

to a downward spiral, thus affecting the team's agility in executing a sprint and moving forward quickly.

To better understand the strength of dependencies, further studies can build on the provided results and explore the relationships between different problem areas in greater depth. While Scrum belongs to the domain of computer science, the results could open research for other domains. Especially the challenge of Scrum teams being embedded in waterfall-ish environments should be further investigated by occupational and organizational psychology researchers to shine more light on the influence of a company's working setting and the resulting performance of an agile software development team.

Based on the broad range of identified issues and given the fact that Scrum teams utilize agile ALM tools for managing their daily work, Chapter 7 aimed at exploring the connection between Scrum problems and tools by investigating the second research question:

"What is the status quo of Scrum tool support?"

As not all programs available on the market could be examined, the analysis focused on four agile project management applications, two market leaders, and two other programs that explicitly advertise their Scrum-specific feature set. By analyzing these four tools in the field and investigating their usability through heuristic evaluation, this thesis clearly illustrates the capabilities and boundaries of today's Scrum tools because all investigated applications share the same problems.

Besides general usability issues, which were shown to affect teams working methods, the most profound outcome is that the investigated applications lack core elements of the Scrum framework, such as acceptance criteria, the definition of ready and definition of done, or even the sprint goal, and beyond that, to some degree, even contradict the obligatory rule set.

What is more, the investigated applications showed severe limitations for cooperative work carried out during the sprint cycle events. The lack of dedicated support for the daily Scrum, sprint review, and sprint retrospective means that essential parts of the Scrum framework are being ignored. However, these are the cornerstones of Scrum's built-in empirical process control, which serves to inspect progress and adapt work to achieve a mutual (sprint) goal. Hence, it must be concluded that the limitations of the investigated tools are not only of a minor nature but must be considered serious deficiencies.

Given the fact that these programs are used on a daily basis and are, therefore, an essential part of the software development process, it must also be concluded that today's agile ALM tools not only relate but contribute to the previously identified Scrum challenges and issues, particularly because many of these issues were also identified

in the area of the collaborative Scrum meetings, which are barely supported by the investigated applications. The lack of central elements of the Scrum process further strengthens the suspicion that the relationship between project management applications and Scrum problems could be closer than expected. While this relationship has not been sufficiently investigated before, further research could take the provided results as a starting point to shed more light on this connection and better understand the influence of software tools on agile development principles.

Besides missing features, the field studies of this thesis further showed that today's agile project management applications are naturally limited because of their underlying interaction paradigm. Graphical User Interfaces (GUIs) relying on mouse or keyboard interactions are designed from the bottom up to be operated by a single user on a notebook or desktop computer. As a result, they are constrained in their ability to support co-local collaborative activities, which demand smooth and simultaneous multi-user interactions to foster mutual discussions and decision-making.

Considering the results that today's agile project management applications are not tailored to the special demands of Scrum teams, the purpose of the final part of this thesis was two-fold. First, it aimed at developing a novel agile management tool that particularly addresses the previously identified Scum issues, thus providing dedicated features for all the framework's components. Second, it aimed to investigate the new human-computer interaction paradigm of Natural User Interfaces (NUIs), known to suit collaborative environments better because, in contrast to GUIs, NUIs do not rely on single points of focus (via mouse pointers or keyboards) but can process multiple inputs simultaneously (via touch, speech, or gestures), thus allowing multiple users to interact at the same time. Therefore, Part III focused on the third research question, asking:

"What could a novel Scrum tool look like utilizing NUI technologies for collaborative activities?"

To answer this question, Chapter 8 first presented various NUI types and compared speech, gestural, and touch interfaces against their individual benefits and weaknesses. By deriving important design considerations for integrating NUIs into the Scrum process and concluding that touch-based NUIs are best suited to enhance collaborative Scrum activities because of their ability to adapt to different tasks and their versatility of interaction devices, this thesis laid the ground for future work in this relatively unexplored research domain.

While existing NUI systems do not exceed the state of being rudimentary research prototypes and hence show significant limitations for practical use, e.g., because of unreliable touch point processing,

poor interface designs causing bad usability, and the lack of essential Scrum-related features, the final part of this thesis strived for higher standards and aimed for developing a full-fledged Scrum management tool that addresses the previously identified Scrum problems.

As a result, Chapter 9 presented an "interactive Scrum space" consisting of desktop, mobile, vertical display, and tabletop interfaces, all being connected through the central *edelsprint* application. For the first time, this novel approach combines both interaction paradigms of classical desktop GUIs and modern touch-based NUI technologies into an *integrated solution* carefully designed for both single-user and collaborative activities within the Scrum framework and taking into account the individual strengths and weaknesses resulting from different form factors of the interaction devices.

Thanks to its strict structure according to the Scrum rules and the dedicated support for the entire framework, it was shown that the developed application addresses 31 of the 42 identified Scrum problems. This must be treated as a very positive result since the remaining 11 problems are hardly manageable by software at all but primarily result from former mismanagement of companies, leaving no room for agility, and hence should be targeted by change management to optimize internal work processes.

Furthermore, it must be considered a success that the developed software went beyond the scope of a research prototype and was brought to market, where it received excellent feedback from both users and industry experts. Nonetheless, it is worth mentioning that these positive outcomes do not automatically imply long-term commercial success of the application because of the sheer dominance of today's market leaders and the fact that companies are pretty resistant to changing their software tools.

However, without further notes about future business strategies, which might also include open-sourcing, this thesis presented a novel full-fledged agile project management application covering the entire Scrum framework, including dedicated features to support all of its obligatory meetings. This achievement is especially mentioned because not only does the software surpass the features of existing Scrum tools, but the dedicated meeting support following the sprint cycle rules also opens exciting new possibilities for unseen research.

As an example, supporting all facets of the framework, including all activities of the sprint cycle, lays the ground for enabling a project management application to capture all sorts of process-related data, including but certainly not limited to the following examples:

- checking whether meetings have been conducted at the same time and calculating conduction rates,
- checking whether meetings exceed the timebox on a regular basis and calculating exceedance rates,
- checking the meeting attendance and calculating show and no-show rates of participants,
- checking whether new improvements are derived from the sprint retrospective and if at least one process improvement is included in the sprint,
- checking whether all participants of the daily Scrum share their work progress on a regular basis,
- checking whether daily Scrums lead to new shared task assignments (indicating willingness to help others),
- checking whether items are mutually estimated during the sprint planning event and calculating the average estimation time.

Future research could build upon this foundation to derive new insights from analyzing data that is now available for the first time. For example, it is conceivable to use AI technology to analyze a team's collected sprint data to measure its performance and automatically identify improvement potentials for a better mastering of Scrum.

Leveraging AI potentials could furthermore derive quality measures of proper backlog management, e.g., by analyzing the formulation of acceptance criteria, deriving correlation with finish rates of user stories, or investigating relationships between backlog items and their positions in the backlog.

All of these are interesting examples of what could be possible in the near future, given that project management applications start to collect the necessary data, thus enabling AI technologies to analyze the software development process.

Overall, the ultimate purpose of this thesis was to investigate the issues of Scrum and contribute to a better mastery of the most widely used software development framework. With the insights and results presented, this goal now should be one step closer.

BIBLIOGRAPHY

[1] Christopher Ackad et al. "Seamless and continuous user identification for interactive tabletops using personal device handshaking and body tracking." In: *Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts - CHI EA '12*. Austin, Texas, USA: ACM Press, 2012, p. 1775. URL: https://doi.org/10.1145/2212776.2223708.

[2] Nao Akechi, Tsukasa Mizumata, and Ryuuki Sakamoto. "Hovering fingertips detection on diffused surface illumination." In: *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*. ITS '11. New York, NY, USA: Association for Computing Machinery, Nov. 2011. URL: https://doi.org/10.1145/2076354.2076422.

[3] Agile Alliance. *What is Role-Feature-Reason?* Dec. 2015. URL: https://www.agilealliance.org/glossary/role-feature/ (visited on 01/22/2019).

[4] Agile Alliance. *Advancing the Practice of Agile*. June 2019. URL: https://www.agilealliance.org/ (visited on 11/09/2019).

[5] Agile Alliance. *What are Story Points? | Agile Alliance*. 2019. URL: https://www.agilealliance.org/glossary/points-estimates-in/ (visited on 12/30/2019).

[6] Ann Anderson et al. "Chrysler goes to "extremes"." In: *Distributed computing* 1.10 (1998), pp. 24–28. URL: https://www.cs.hmc.edu/courses/2004/spring/cs121/papers/xpChrysler.pdf.

[7] Samuil Angelov, Marcel Meesters, and Matthias Galster. "Architects in Scrum: What Challenges Do They Face?" In: *Software Architecture*. Ed. by Bedir Tekinerdogan, Uwe Zdun, and Ali Babar. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 229–237. URL: https://doi.org/10.1007/978-3-319-48992-6_17.

[8] Anoto AB. *Bridging the anaolog and digital divide*. 2024. URL: https://www.anoto.com/ (visited on 05/20/2024).

[9] Hal R. Arkes and Catherine Blumer. "The psychology of sunk cost." In: *Organizational Behavior and Human Decision Processes* 35.1 (1985), pp. 124–140. URL: https://doi.org/10.1016/0749-5978(85)90049-4.

[10] John Armitage. "Are Agile Methods Good for Design?" In: *Interactions* 11.1 (Jan. 2004), pp. 14–23. URL: https://doi.org/10.1145/962342.962352.

[11] Barry Arons. "SpeechSkimmer: a system for interactively skimming recorded speech." In: *ACM Transactions on Computer-Human Interaction* 4.1 (Mar. 1997), pp. 3–38. URL: https://doi.org/10.1145/244754.244758.

[12] Stan Augarten. *Bit by Bit: An Illustrated History of Computers*. New York: Houghton Mifflin Harcourt, Nov. 1984.

[13] Matthew P. Aylett, Benjamin R. Cowan, and Leigh Clark. "Siri, Echo and Performance: You have to Suffer Darling." In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI EA '19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–10. URL: https://doi.org/10.1145/3290607.3310422.

[14] Stefan Bachl et al. "Challenges for Designing the User Experience of Multi-touch interfaces." In: vol. Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems. Berlin, EU: ACM, Jan. 2010. URL: http://hdl.handle.net/20.500.12708/53538.

[15] Julian M. Bass et al. "An empirical study of the product owner role in scrum." In: *Proceedings of the 40th international conference on software engineering: Companion proceeedings*. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 123–124. URL: https://doi.org/10.1145/3183440.3195066.

[16] K. Beck. "Embracing change with extreme programming." In: *Computer* 32.10 (Oct. 1999), pp. 70–77. URL: https://doi.org/10.1109/2.796139.

[17] Kent Beck. "Extreme programming: A humanistic discipline of software development." In: *Fundamental Approaches to Software Engineering*. Ed. by Egidio Astesiano. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 1–6. URL: https://doi.org/10.1007/BFb0053579.

[18] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. 2nd edition. Boston: Addison-Wesley Professional, Nov. 2004.

[19] Kent Beck et al. *Manifesto for agile software development*. 2001. URL: http://www.agilemanifesto.org/.

[20] Jeff Beckman. *The Most Critical Scrum Usage Statistics [2023 Edition]*. Aug. 2023. URL: https://techreport.com/statistics/scrum-usage-statistics/ (visited on 02/24/2024).

[21] T. E. Bell and T. A. Thayer. "Software Requirements: Are They Really a Problem?" In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 61–68. URL: https://dl.acm.org/doi/10.5555/800253.807650.

[22] H. D. Benington. "Production of large computer programs." In: *Proceedings of the 9th international conference on software engineering*. Icse '87. Washington, DC, USA: IEEE Computer Society Press, 1987, pp. 299–310. URL: https://dl.acm.org/doi/10.5555/41765.41799.

[23] David Benyon. *Designing Interactive Systems: A Comprehensive Guide to HCI and Interaction Design*. 2nd ed. Harlow Munich: Pearson Education Limited, Apr. 2010.

[24] Andrea Bianchi and Seungwoo Je. "Disambiguating touch with a smart-ring." In: *Proceedings of the 8th Augmented Human International Conference*. AH '17. New York, NY, USA: Association for Computing Machinery, Mar. 2017, pp. 1–5. URL: https://doi.org/10.1145/3041164.3041196.

[25] Olga Blinova. *Release Planning as long-term vision in Scrum*. Master Thesis. Paderborn University, 2017.

[26] Barry W. Boehm. "Software Engineering." In: *IEEE Trans. Comput.* 25.12 (Dec. 1976), pp. 1226–1241. URL: https://doi.org/10.1109/TC.1976.1674590.

[27] Barry W. Boehm. "Guidelines for Verifying and Validating Software Requirements and Design Specifications." In: *Euro IFIP 79*. North Holland, 1979, p. 20.

[28] Barry W. Boehm. *Software Engineering Economics*. Englewood Cliffs, N.J: Prentice-Hall, 1981.

[29] Barry. W. Boehm. "Verifying and Validating Software Requirements and Design Specifications." In: *IEEE Softw.* 1.1 (Jan. 1984), pp. 75–88. URL: https://doi.org/10.1109/MS.1984.233702.

[30] Barry W. Boehm. "A Spiral Model of Software Development and Enhancement." In: *ACM SIGSOFT Software Engineering Notes* 11.4 (Aug. 1986), pp. 14–24. URL: https://doi.org/10.1145/12944.12948.

[31] Barry W. Boehm. "A spiral model of software development and enhancement." In: *IEEE Computer* 21.5 (May 1988), pp. 61–72. URL: https://doi.org/10.1109/2.59 (visited on 12/21/2018).

[32] Barry W. Boehm. "A view of 20th and 21st century software engineering." In: *Proceeding of the 28th international conference on Software engineering - ICSE '06*. Shanghai, China: ACM Press, 2006, p. 12. URL: https://doi.org/10.1145/1134285.1134288.

[33] Barry W. Boehm and P. Bose. "A collaborative spiral software process model based on Theory W." In: *Proceedings of the Third International Conference on the Software Process. Applying the Software Process*. Reston, VA, USA: IEEE Comput. Soc. Press, 1994, pp. 59–68. URL: https://doi.org/10.1109/SPCON.1994.344423.

[34] Barry W. Boehm and Wilfred J. Hansen. *Spiral Development: Experience, Principles, and Refinements:* tech. rep. Fort Belvoir, VA: Defense Technical Information Center, July 2000. URL: http://www.dtic.mil/docs/citations/ADA382590.

[35] Barry W. Boehm et al. "Using the WinWin Spiral Model: A Case Study." In: *Computer* 31.7 (July 1998), pp. 33–44. URL: https://doi.org/10.1109/2.689675.

[36] Roy Boggs. "The sdlc and six sigma - an essay on which is which and why?" In: *Issues in Information Systems* V.1 (2004), pp. 36–42. URL: https://www.researchgate.net/publication/288009904_The_SDLC_and_Six_Sigma_An_Essay_on_Which_is_Which_and_Why.

[37] John G. Brainerd. "Genesis of the ENIAC." In: *Technology and Culture* 17.3 (July 1976), p. 482. URL: https://doi.org/10.2307/3103527.

[38] Brooks. "No Silver Bullet Essence and Accidents of Software Engineering." In: *Computer* 20.4 (Apr. 1987), pp. 10–19. URL: https://doi.org/10.1109/MC.1987.1663532.

[39] Frederick P. Brooks. *The mythical man-month - essays on software engineering (2. ed.)* Addison-Wesley, 1995.

[40] Manfred Broy. "Yesterday, Today, and Tomorrow: 50 Years of Software Engineering." In: *IEEE Software* 35.5 (Sept. 2018), pp. 38–43. URL: https://doi.org/10.1109/MS.2018.290111138.

[41] Antony Bryant. "'It's Engineering Jim ... but not as we know it' - Software Engineering - solution to the software crisis, or part of the problem?" In: *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. 2000, pp. 78–87. URL: https://doi.org/10.1145/337180.337191.

[42] Roger Buehler, Dale Griffin, and Michael Ross. "Exploring the "Planning Fallacy": Why People Underestimate Their Task Completion Times." In: *Journal of Personality and Social Psychology* 67.3 (1994), pp. 366–381. URL: https://doi.org/10.1037/0022-3514.67.3.366.

[43] Roger Buehler, Dale Griffin, and Michael Ross. "Inside the planning fallacy: The causes and consequences of optimistic time predictions." In: *Heuristics and biases: The psychology of intuitive judgment*. New York, NY, US: Cambridge University

Press, 2002, pp. 250–270. URL: https://doi.org/10.1017/CBO9780511808098.016.

[44] Roger Buehler, Deanna Messervey, and Dale Griffin. "Collaborative planning and prediction: Does group discussion affect optimistic biases in time estimation?" In: *Organizational Behavior and Human Decision Processes* 97.1 (May 2005), pp. 47–63. URL: https://doi.org/10.1016/j.obhdp.2005.02.004.

[45] Bill Buxton. *Multi-Touch Systems that I Have Known and Loved*. May 2020. URL: http://billbuxton.com/multitouchOverview.html (visited on 03/11/2021).

[46] François Bérard and Yann Laurillau. "Single user multitouch on the DiamondTouch: from 2 x 1D to 2D." In: *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*. ITS '09. New York, NY, USA: Association for Computing Machinery, Nov. 2009, pp. 1–8. URL: https://doi.org/10.1145/1731903.1731905.

[47] Marcio C. Cabral, Carlos H. Morimoto, and Marcelo K. Zuffo. "On the usability of gesture interfaces in virtual reality environments." In: *Proceedings of the 2005 Latin American conference on Human-computer interaction*. CLIHC '05. New York, NY, USA: Association for Computing Machinery, Oct. 2005, pp. 100–108. URL: https://doi.org/10.1145/1111360.1111370.

[48] T. Carlyle and E. Markham. *On heroes, hero-worship, and the heroic in history: Six lectures, reported, with emendations and additions*. D. Appleton & Company, 1842. URL: https://books.google.de/books?id=wNA5AQAAMAAJ.

[49] E. Carmel, J. F. George, and J. F. Nunamaker. "Supporting joint application development (JAD) and electronic meeting systems: moving the CASE concept into new areas of software development." In: *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*. Vol. iii. Jan. 1992, 331–342 vol.3. URL: https://doi.org/10.1109/HICSS.1992.183501.

[50] Gavin Cassar. "Are individuals entering self-employment overly optimistic? an empirical test of plans and projections on nascent entrepreneur expectations." In: *Strategic Management Journal* 31.8 (2010), pp. 822–840. URL: https://doi.org/10.1002/smj.833.

[51] Elisa Marques de Castro and Luciana Martinez Zaina. "Investigating the interaction of children through NUI in e-learning applications." In: *Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems*. IHC 2017. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1–10. URL: https://doi.org/10.1145/3160504.3160519.

[52] Juyun Cho. "Issues and challenges of agile software development with Scrum." In: *Issues in Information Systems* 9.2 (2008), p. 8. URL: https://doi.org/10.48009/2_iis_2008_188-195.

[53] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley Professional, Oct. 2004.

[54] Alistair Cockburn. "Using Both Incremental and Iterative Development." In: *STSC CrossTalk (USAF Software Technology Support Center)* 21.5 (2008), pp. 27–30.

[55] Mike Cohn. *User Stories Applied: For Agile Software Development*. 1st ed. Boston: Addison-Wesley Professional, Mar. 2004.

[56] Mike Cohn. *Agile Estimating and Planning*. 1st ed. Upper Saddle River, NJ: Prentice Hall, Nov. 2005.

[57] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Pearson Education, 2010.

[58] Mike Cohn. *Agile Excel Spreadsheet for the Product Backlog*. July 2011. URL: https://www.mountaingoatsoftware.com/blog/a-sample-format-for-a-spreadsheet-based-product-backlog (visited on 03/22/2020).

[59] Mike Cohn. *Product Backlog Refinement*. May 2015. URL: https://www.mountaingoatsoftware.com/blog/product-backlog-refinement-grooming (visited on 01/20/2024).

[60] Mike Cohn. *Product backlog refinement (grooming)*. 2015. URL: http://athena.ecs.csus.edu/~buckley/CSc233/Backlog_Grooming.pdf (visited on 12/12/2020).

[61] Mike Cohn. *Planning Poker: An Agile Estimating and Planning Technique*. 2020. URL: https://www.mountaingoatsoftware.com/agile/planning-poker (visited on 01/15/2020).

[62] DSDM Consortium. *DSDM: Business Focused Development*. Pearson Education, 2003.

[63] Eric Corbett and Astrid Weber. "What can I say? addressing user experience challenges of a mobile voice user interface for accessibility." In: *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services*. MobileHCI '16. New York, NY, USA: Association for Computing Machinery, Sept. 2016, pp. 72–82. URL: https://doi.org/10.1145/2935334.2935386.

[64] José Adson Oliveira Guedes da Cunha and Hermano Perrelli de Moura. "Towards a substantive theory of project decisions in software development project-based organizations: A cross-case analysis of IT organizations from Brazil and Portugal." In: *2015 10th Iberian Conference on Information Systems and Tech-*

nologies (CISTI). June 2015, pp. 1–6. URL: https://doi.org/10.1109/CISTI.2015.7170515.

[65] M. A. Cusumano. "The software factory: a historical interpretation." In: *IEEE Software* 6.2 (Mar. 1989), pp. 23–30. URL: https://doi.org/10.1109/MS.1989.1430446.

[66] Raimund Dachselt and Robert Buchholz. "Natural throw and tilt interaction between mobile phones and distant displays." In: *CHI '09 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '09. New York, NY, USA: Association for Computing Machinery, Apr. 2009, pp. 3253–3258. URL: https://doi.org/10.1145/1520340.1520467.

[67] Bo Dahlbom. "The New Informatics." In: *Scandinavian Journal of Information Systems* 8.2 (Jan. 1996). URL: https://aisel.aisnet.org/sjis/vol8/iss2/3.

[68] Andries van Dam. "Post-WIMP user interfaces." In: *Communications of the ACM* 40.2 (Feb. 1997), pp. 63–67. URL: https://doi.org/10.1145/253671.253708.

[69] Lorraine Daston. "Enlightenment Calculations." In: *Critical Inquiry* 21.1 (Oct. 1994), pp. 182–202. URL: https://www.journals.uchicago.edu/doi/10.1086/448745.

[70] Rachel Davies. *The Power of Stories*. 2001. URL: https://www.researchgate.net/publication/2532068_The_Power_of_Stories.

[71] Noopur Davis. "Driving Quality Improvement and Reducing Technical Debt with the Definition of Done." In: *2013 Agile Conference*. Nashville, TN, USA: IEEE, Aug. 2013, pp. 164–168. URL: https://doi.org/10.1109/AGILE.2013.21.

[72] Li Deng and Xuedong Huang. "Challenges in adopting speech recognition." In: *Communications of the ACM* 47.1 (Jan. 2004), pp. 69–75. URL: https://doi.org/10.1145/962081.962108.

[73] Esther Derby and Diana Larsen. *Agile Retrospectives: Making Good Teams Great*. 1st ed. Raleigh, NC: O'Reilly UK Ltd., Aug. 2006.

[74] Paul Dietz and Darren Leigh. "DiamondTouch: a multi-user touch technology." In: *Proceedings of the 14th annual ACM symposium on User interface software and technology*. UIST '01. New York, NY, USA: Association for Computing Machinery, Nov. 2001, pp. 219–226. URL: https://doi.org/10.1145/502348.502389.

[75] Edsger W. Dijkstra. "Letters to the Editor: Go to Statement Considered Harmful." In: *Commun. ACM* 11.3 (Mar. 1968), pp. 147–148. URL: https://doi.org/10.1145/362929.362947.

[76] Michal Doležel and Michael Felderer. "Organizational Patterns between Developers and Testers - Investigating Testers' Autonomy and Role Identity:" in: *Proceedings of the 20th International Conference on Enterprise Information Systems*. Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pp. 336–344. URL: https://doi.org/10.5220/0006783703360344.

[77] Paul Dourish. *Where the Action Is: The Foundations of Embodied Interaction*. Cambridge, Mass: MIT Press, Nov. 2001.

[78] Pierre Dragicevic and Yuanchun Shi. "Visualizing and manipulating automatic document orientation methods using vector fields." In: *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*. ITS '09. New York, NY, USA: Association for Computing Machinery, Nov. 2009, pp. 65–68. URL: https://doi.org/10.1145/1731903.1731918.

[79] John Dumay and Sandy Q. Qu. "The qualitative research interview." In: *Qualitative Research in Accounting & Management* 8.3 (Aug. 2011), pp. 238–264. URL: https://doi.org/10.1108/11766091111162070.

[80] M. D. Dunlop and M. Montgomery Masters. "Pickup usability dominates: a brief history of mobile text entry research and adoption." In: *International Journal of Mobile Human Computer Interaction* 1.1 (2009), pp. 42–59. URL: https://doi.org/10.4018/jmhci.2009010103.

[81] Douglas C. Engelbart and William K. English. "A Research Center for Augmenting Human Intellect." In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 395–410. URL: https://doi.org/10.1145/1476589.1476645.

[82] Morten Esbensen et al. "The dBoard: A Digital Scrum Board for Distributed Software Development." In: *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces - ITS '15*. Madeira, Portugal: ACM Press, 2015, pp. 161–170. URL: https://doi.org/10.1145/2817721.2817746.

[83] Jinjuan Feng et al. "Speech-based navigation and error correction: a comprehensive comparison of two solutions." In: *Universal Access in the Information Society* 10.1 (Mar. 2011), pp. 17–31. URL: https://doi.org/10.1007/s10209-010-0185-9.

[84] Therese Fessenden. *Net Promoter Score: What a Customer-Relations Metric Can Tell You About Your User Experience*. Oct. 2016. URL: https://www.nngroup.com/articles/nps-ux/ (visited on 08/13/2023).

[85] Clifton Forlines et al. "Direct-touch vs. mouse input for tabletop displays." In: *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '07*. San Jose, California, USA: ACM Press, 2007, p. 647. URL: https://doi.org/10.1145/1240624.1240726.

[86] Martin Fowler. "The new methodology." In: *Wuhan University Journal of Natural Sciences* 6.1-2 (Mar. 2001), pp. 12–24. URL: https://doi.org/10.1007/BF03160222.

[87] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd ed. Addison Wesley, Jan. 2019.

[88] Euan Freeman, Stephen Brewster, and Vuokko Lantz. "Tactile Feedback for Above-Device Gesture Interfaces: Adding Touch to Touchless Interactions." In: *Proceedings of the 16th International Conference on Multimodal Interaction*. ICMI '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 419–426. URL: https://doi.org/10.1145/2663204.2663280.

[89] W. Barkley Fritz. "The Women of ENIAC." In: *IEEE Ann. Hist. Comput.* 18.3 (Sept. 1996), pp. 13–28. URL: https://doi.org/10.1109/85.511940.

[90] Taghi Javdani Gandomani. "A Case Study Research on Software Cost Estimation Using Experts' Estimates, Wideband Delphi, and Planning Poker Technique." In: *International Journal of Software Engineering and Its Applications* 8.11 (2014), pp. 173–182. URL: https://api.semanticscholar.org/CorpusID:11128442.

[91] Andreas Gehle. *Konzept und prototypische Implementierung einer digitalen Unterstützung von Sprint-Retrospektiven in Scrum*. Master Thesis. Paderborn University, 2017.

[92] E Georgiadou. "Software Process and Product Improvement: A Historical Perspective." In: *Cybernetics and Systems Analysis*. 39 1. Plenum Publishing Corporation, 2003, pp. 125–142. URL: https://doi.org/10.1023/A:1023833428613.

[93] Sebastian Gerhardt. *Supporting Decision-Making in Agile Development*. Master Thesis. Paderborn University, 2018.

[94] Yaser Ghanam, Xin Wang, and Frank Maurer. "Utilizing Digital Tabletops in Collocated Agile Planning Meetings." In: *Agile 2008 Conference*. Toronto, ON, Canada: IEEE, 2008, pp. 51–62. URL: https://doi.org/10.1109/Agile.2008.13.

[95] Tom Gilb. "Evolutionary Delivery Versus the "Waterfall Model"." In: *SIGSOFT Softw. Eng. Notes* 10.3 (July 1985), pp. 49–61. URL: https://doi.org/10.1145/1012483.1012490.

[96] Gerard Goggin. "Adapting the mobile phone: The iPhone and its consumption." In: *Continuum* 23.2 (Apr. 2009), pp. 231–244. URL: https://doi.org/10.1080/10304310802710546.

[97] Beverly E. Golemba. "Human Computers: The Women in Aeronautical Research." In: *Unpublished* (1995). URL: http://crgis.ndc.nasa.gov/crgis/images/c/c7/Golemba.pdf.

[98] James Grenning. "Planning Poker or How to avoid analysis paralysis while release planning." In: *Hawthorn Woods: Renaissance Software Consulting* 3 (2002).

[99] David Alan Grier. "The ENIAC, the verb "to program" and the emergence of digital computers." In: *IEEE Annals of the History of Computing* 18.1 (1996), pp. 51–55. URL: https://doi.org/10.1109/85.476561.

[100] David Alan Grier. "Gertrude Blanch of the Mathematical Tables Project." In: *IEEE Annals of the History of Computing* 19.4 (Oct. 1997), pp. 18–27. URL: https://doi.org/10.1109/85.627896.

[101] David Alan Grier. "The Math Tables Project of the work projects administration: the reluctant start of the computing era." In: *IEEE Annals of the History of Computing* 20.3 (Sept. 1998), pp. 33–50. URL: https://doi.org/10.1109/85.707573.

[102] David Alan Grier. "Human computers: the first pioneers of the information age." In: *Endeavour* 25.1 (Mar. 2001), pp. 28–32. URL: https://doi.org/10.1016/S0160-9327(00)01338-7.

[103] David Alan Grier. *When Computers Were Human*. Princeton University Press, 2005.

[104] Isaac Griffith et al. "A simulation study of practical methods for technical debt management in agile software development." In: *Proceedings of the Winter Simulation Conference 2014*. Savanah, GA, USA: IEEE, Dec. 2014, pp. 1014–1025. URL: https://doi.org/10.1109/WSC.2014.7019961.

[105] The Standish Group. *The CHAOS Report (1994)*. Tech. rep. 1994. URL: https://www.standishgroup.com/sample_research_files/chaos_report_1994.pdf (visited on 01/07/2019).

[106] Klaus G. Grunert and Suzanne C. Grunert. "Measuring subjective meaning structures by the laddering method: Theoretical considerations and methodological problems." In: *International Journal of Research in Marketing* 12.3 (Oct. 1995), pp. 209–225. URL: https://doi.org/10.1016/0167-8116(95)00022-T.

[107] Denise Gürer. "Women in computing history." In: *ACM SIGCSE Bulletin* 34.2 (June 2002), p. 116. URL: https://doi.org/10.1145/543812.543843.

[108] Michael Haller et al. *Interactive Displays and Next-Generation Interfaces*. Hagenberg Research. Springer-Verlag Berlin Heidelberg, 2009.

[109] Edmund Halley. *A Synopsis of the Astronomy of Comets*. Tech. rep. Printed for John Senex, 1705, p. 58.

[110] Jeff Han. *The radical promise of the multi-touch interface*. Feb. 2006. URL: https://www.ted.com/talks/jeff_han_the_radical_promise_of_the_multi_touch_interface (visited on 03/15/2021).

[111] Jefferson Y. Han. "Low-cost multi-touch sensing through frustrated total internal reflection." In: *Proceedings of the 18th annual ACM symposium on User interface software and technology - UIST '05*. Seattle, WA, USA: ACM Press, 2005, p. 115. URL: https://doi.org/10.1145/1095034.1095054.

[112] Zaliyana Mohd Hanafiah et al. "Human-robot speech interface understanding inexplicit utterances using vision." In: *CHI '04 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '04. New York, NY, USA: Association for Computing Machinery, Apr. 2004, pp. 1321–1324. URL: https://doi.org/10.1145/985921.986054.

[113] J. Haungs. "Pair programming on the C3 project." In: *Computer* 34.2 (Feb. 2001), pp. 118–119. URL: https://doi.org/10.1109/2.901173.

[114] Ville T. Heikkilä, Maria Paasivaara, and Casper Lassenius. "ScrumBut, But Does it Matter? A Mixed-Method Study of the Planning Process of a Multi-team Scrum Organization." In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. Oct. 2013, pp. 85–94. URL: https://doi.org/10.1109/ESEM.2013.27.

[115] P. Herzlich. "RAD and quality principles." In: *IEE Colloquium on Will Tickit and ISO 9000 Survive Rapid Application Development?* Dec. 1995, pp. 2/1–2/5. URL: https://doi.org/10.1049/ic:19951553.

[116] James A. Highsmith. *Adaptive Software Development: An Evolutionary Approach to Controlling Chaotic Systems: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House Publishing Co Inc.,U.S., Dec. 1999.

[117] Juan David Hincapié-Ramos et al. "Consumed endurance: a metric to quantify arm fatigue of mid-air interactions." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 1063–1072. URL: https://doi.org/10.1145/2556288.2557130.

[118] Christian Holz and Patrick Baudisch. "Fiberio: a touchscreen that senses fingerprints." In: *Proceedings of the 26th annual ACM symposium on User interface software and technology*. UIST '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 41–50. URL: https://doi.org/10.1145/2501988.2502021.

[119] Andrew Hunt, David Thomas, and Ward Cunningham. *The Pragmatic Programmer. From Journeyman to Master*. 1st ed. Reading, Mass: Addison Wesley, Oct. 1999.

[120] Adrian Hülsmann and Julian Maicher. "HOUDINI: Introducing Object Tracking and Pen Recognition for LLP Tabletops." In: *Human-Computer Interaction. Advanced Interaction Modalities and Techniques*. Ed. by David Hutchison et al. Vol. 8511. Cham: Springer International Publishing, 2014, pp. 234–244. URL: https://doi.org/10.1007/978-3-319-07230-2_23.

[121] M. Imaz and D. Benyon. "How Stories Capture Interactions." In: *IFIP TC13 International Conference on Human-Computer Interaction*. 1999. URL: https://api.semanticscholar.org/CorpusID:5508979.

[122] CASEMaker Inc. "What is Rapid Application Development (RAD)." In: *Unpublished*. 1997. URL: https://www.iro.umontreal.ca/~dift6803/Transparents/Chapitre1/Documents/rad_wp.pdf (visited on 06/21/2019).

[123] Hiroshi Ishii. "The tangible user interface and its evolution." In: *Communications of the ACM* 51.6 (June 2008), pp. 32–36. URL: https://doi.org/10.1145/1349026.1349034.

[124] Hiroshi Ishii and Brygg Ullmer. "Tangible bits: towards seamless interfaces between people, bits and atoms." In: *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '97*. Atlanta, Georgia, United States: ACM Press, 1997, pp. 234–241. URL: https://doi.org/10.1145/258549.258715.

[125] Hiroshi Ishii et al. "Radical atoms: beyond tangible bits, toward transformable materials." In: *Interactions* 19.1 (Jan. 2012), pp. 38–51. URL: https://doi.org/10.1145/2065327.2065337.

[126] Jhilmil Jain, Arnold Lund, and Dennis Wixon. "The future of natural user interfaces." In: *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems - CHI EA '11*. Vancouver, BC, Canada: ACM Press, 2011, p. 211. URL: https://doi.org/10.1145/1979742.1979527.

[127] Jakob Nielsen. *Thinking Aloud: The #1 Usability Tool*. Jan. 2012. URL: https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/ (visited on 05/27/2024).

[128] Mikkel R. Jakobsen and Kasper HornbÆk. "Up close and personal: Collaborative work on a high-resolution multitouch wall display." In: *ACM Transactions on Computer-Human Interaction* 21.2 (Feb. 2014), 11:1–11:34. URL: https://doi.org/10.1145/2576099.

[129] Ron Jeffries. *We'll Try*. July 1999. URL: https://ronjeffries.com/xprog/articles/well_try/ (visited on 01/11/2020).

[130] Ron Jeffries. *Essential XP: Card, Conversation, Confirmation*. Aug. 2001. URL: http://ronjeffries.com/xprog/articles/ (visited on 01/23/2019).

[131] Hans Jetter, Harald Reiterer, and Florian Geyer. "Blended Interaction: understanding natural human-computer interaction in post-WIMP interactive spaces." In: *Personal and Ubiquitous Computing* 18.5 (June 2014), pp. 1139–1158. URL: https://doi.org/10.1007/s00779-013-0725-4.

[132] Sergi Jordà. "The reactable: tangible and tabletop music performance." In: *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems - CHI EA '10*. Atlanta, Georgia, USA: ACM Press, 2010, p. 2989. URL: https://doi.org/10.1145/1753846.1753903.

[133] D. Kahneman and A. Tversky. "Intuitive Prediction: Biases and Corrective Procedures." In: *TIMS Studies in Management Science* 12 (1979), pp. 313–327. URL: https://apps.dtic.mil/sti/citations/ADA047747.

[134] Raine Kajastila and Tapio Lokki. "Eyes-free interaction with free-hand gestures and auditory menus." In: *International Journal of Human-Computer Studies* 71.5 (May 2013), pp. 627–640. URL: https://doi.org/10.1016/j.ijhcs.2012.11.003.

[135] Martin Kaltenbrunner. "reacTIVision and TUIO: a tangible tabletop toolkit." In: *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces - ITS '09*. Banff, Alberta, Canada: ACM Press, 2009, p. 9. URL: http://portal.acm.org/citation.cfm?doid=1731903.1731906 (visited on 11/17/2018).

[136] Martin Kaltenbrunner and Ross Bencina. "reacTIVision: a computer-vision framework for table-based tangible interaction." In: *Proceedings of the 1st international conference on Tangible and embedded interaction - TEI '07*. Baton Rouge, Louisiana: ACM Press, 2007, p. 69. URL: https://doi.org/10.1145/1226969.1226983.

[137] Frederic Kaplan. "Are gesture-based interfaces the future of human computer interaction?" In: *Proceedings of the 2009 international conference on Multimodal interfaces*. ICMI-MLMI '09. New York, NY, USA: Association for Computing Machinery, Nov. 2009, pp. 239–240. URL: https://doi.org/10.1145/1647314.1647365.

[138] Kendra Cherry. *What Are the Gestalt Principles?* Mar. 2023. URL: https://www.verywellmind.com/gestalt-laws-of-perceptual-organization-2795835 (visited on 04/11/2024).

[139] Sam Kinsley. *Addressing ubicomp: Computing people, places and things*. Mar. 2010. URL: http://www.samkinsley.com/2010/03/13/addressing-ubicomp-computing-people-places-and-things/ (visited on 03/05/2021).

[140] Christian Klaussner. *A Virtual Scrum Coach to Improve Agile Process Quality*. Master Thesis. Paderborn University, 2019.

[141] Donald E. Knuth. "Computer programming as an art." In: *Communications of the ACM* 17.12 (Dec. 1974), pp. 667–673. URL: https://doi.org/10.1145/361604.361612.

[142] Cris Kobryn. "UML 2001: A Standardization Odyssey." In: *Commun. ACM* 42.10 (Oct. 1999), pp. 29–37. URL: https://doi.org/10.1145/317665.317673.

[143] A. M. Koss. "Programming on the Univac 1: a woman's account." In: *IEEE Annals of the History of Computing* 25.1 (Jan. 2003), pp. 48–59. URL: https://doi.org/10.1109/MAHC.2003.1179879.

[144] Marek Kołodziejski. *Europäischer Fonds für regionale Entwicklung (EFRE) | Kurzdarstellungen zur Europäischen Union | Europäisches Parlament*. Mar. 2023. URL: https://www.europarl.europa.eu/factsheets/de/sheet/95/europaischer-fonds-fur-regionale-entwicklung-efre- (visited on 07/29/2023).

[145] Iva Krasteva and Sylvia Ilieva. "Adopting an agile methodology: why it did not work." In: *Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral*. APOS '08. Leipzig, Germany: Association for Computing Machinery, May 2008, pp. 33–36. URL: https://doi.org/10.1145/1370143.1370150.

[146] V. Krishna and A. Basu. "Scrum+:: Is it "ScrumBut" or "ScrumAnd"." In: *2011 Annual IEEE India Conference*. Dec. 2011, pp. 1–4. URL: https://doi.org/10.1109/INDCON.2011.6139625.

[147] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[148] Russell Kruger et al. "How people use orientation on tables: comprehension, coordination and communication." In: *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*. GROUP '03. New York, NY, USA: Association for Computing Machinery, Nov. 2003, pp. 369–378. URL: https://doi.org/10.1145/958160.958219.

[149] Russell Kruger et al. "Fluid integration of rotation and translation." In: *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '05*. Portland, Oregon, USA: ACM Press, 2005, p. 601. URL: https://doi.org/10.1145/1054972.1055055.

[150] Eyal Krupka et al. "Toward Realistic Hands Gesture Interface: Keeping it Simple for Developers and Machines." In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 1887–1898. URL: https://doi.org/10.1145/3025453.3025508.

[151] Sven Köhler, Christian Holz, and Patrick Baudisch. "Demonstration and Applications of Fiberio: A Touchscreen That Senses Fingerprints." In: *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*. ITS '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 443–446. URL: https://doi.org/10.1145/2669485.2669530.

[152] Mitch Lacey. *Scrum Field Guide, The: Practical Advice for Your First Year*. 1st ed. Upper Saddle River, NJ: Addison-Wesley Professional, Mar. 2012.

[153] Mitch Lacey. *4 Secrets to a Successful Daily Scrum*. 2018. URL: https://www.mitchlacey.com/blog/4-secrets-to-a-successful-daily-scrum (visited on 05/04/2019).

[154] Butler Lampson. "Personal distributed computing: The alto and ethernet software." In: *A History of Personal Workstations, ed. A. Goldberg, Addison-Wesley* (Jan. 1988), pp. 291–344. URL: https://doi.org/10.1145/61975.66921.

[155] C. Larman and V.R. Basili. "Iterative and incremental developments. a brief history." In: *Computer* 36.6 (June 2003), pp. 47–56. URL: https://doi.org/10.1109/MC.2003.1204375 (visited on 01/10/2019).

[156] Lai-Chong Law and Ebba Thora Hvannberg. "Complementarity and Convergence of Heuristic Evaluation and Usability Test: A Case Study of Universal Brokerage Platform." In: *Proceedings of the Second Nordic Conference on Human-computer Interaction*. NordiCHI '02. New York, NY, USA: ACM, 2002, pp. 71–80. URL: http://doi.acm.org/10.1145/572020.572030.

[157] Beth L. Leech. "Asking Questions: Techniques for Semistructured Interviews." In: *PS: Political Science & Politics* 35.4 (Dec. 2002), pp. 665–668. URL: https://doi.org/10.1017/S1049096502001129.

[158] Dean Leffingwell. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, Jan. 2011.

[159] Jennifer S. Light. "When Computers Were Women." In: *The Johns Hopkins University Press and the Society for the History of Technology* 40.3 (1999), pp. 455–483.

[160] Jun Liu et al. "TNT: improved rotation and translation on digital tables." In: *Proceedings of graphics interface 2006*. Gi '06. CAN: Canadian Information Processing Society, 2006, pp. 25–32. URL: https://dl.acm.org/doi/10.5555/1143079.1143084.

[161] L. Liu, H. Erdogmus, and F. Maurer. "An environment for collaborative iteration planning." In: *Agile Development Conference (ADC'05)*. Denver, CO, USA: IEEE Comput. Soc, 2005, pp. 80–89. URL: https://doi.org/10.1109/ADC.2005.12.

[162] Xiaoxing Liu and Geb W. Thomas. "Gesture Interfaces: Minor Change in Effort, Major Impact on Appeal." In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 4278–4283. URL: https://doi.org/10.1145/3025453.3025513.

[163] Michael S Mahoney. "Finding a History for Software Engineering." In: *IEEE Annals of the History of Computing* 26.1 (2004), pp. 8–19. URL: https://doi.org/0.1109/MAHC.2004.1278847.

[164] Julian Maicher. *Innovative Tool Support for Agile Scrum Teams*. Master Thesis. Paderborn University, 2014.

[165] Antony Marcano. *How the industry broke the Connextra Template | antonymarcano.com*. Aug. 2016. URL: http://antonymarcano.com/blog/2016/08/how-the-industry-broke-the-connextra-template/ (visited on 01/22/2019).

[166] Nicolai Marquardt, Johannes Kiemer, and Saul Greenberg. "What caused that touch?: expressive interaction with a surface through fiduciary-tagged gloves." In: *ACM International Conference on Interactive Tabletops and Surfaces - ITS '10*. Saarbrücken, Germany: ACM Press, 2010, p. 139. URL: https://doi.org/10.1145/1936652.1936680.

[167] Dr C. Dianne Martin. "ENIAC: The Press Conference That Shook the World." In: *IEEE Technology and Society Magazine* 14.4 (2002), pp. 3–10. URL: https://doi.org/10.1109/44.476631.

[168] James Martin. *Rapid Application Development*. Indianapolis, IN, USA: Macmillan Publishing Co., Inc., 1991.

[169] Massachusetts Institute of Technology. *Tangible Media Group*. 2021. URL: https://tangible.media.mit.edu/ (visited on 05/09/2021).

[170] Damien Masson et al. "WhichFingers: Identifying Fingers on Touch Surfaces and Keyboards using Vibration Sensors." In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 41–48. URL: https://doi.org/10.1145/3126594.3126619.

[171] Nobuyuki Matsushita and Jun Rekimoto. "HoloWall: designing a finger, hand, body, and object sensitive wall." In: *Proceedings of the 10th annual ACM symposium on User interface software and technology - UIST '97*. Banff, Alberta, Canada: ACM Press, 1997, pp. 209–210. URL: https://doi.org/10.1145/263407.263549 (visited on 08/18/2022).

[172] John W. Mauchly. "The Use of High Speed Vacuum Tube Devices for Calculating." In: *The Origins of Digital Computers: Selected Papers*. Ed. by Brian Randell. Texts and Monographs in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 355–358. URL: https://doi.org/10.1007/978-3-642-61812-3_28.

[173] Christopher McAdam and Stephen Brewster. "Using Mobile Phones to Interact with Tabletop Computers." In: *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces, ITS'11*. Kobe, Japan: Association for Computing Machinery, Nov. 2011, pp. 232–241. URL: https://doi.org/10.1145/2076354.2076395.

[174] Steve McConnell. *Software Project Survival Guide*. 1 edition. Redmond, Wash: Microsoft Press, Oct. 1997.

[175] Daniel D. McCracken and Michael A. Jackson. "Life Cycle Concept Considered Harmful." In: *SIGSOFT Softw. Eng. Notes* 7.2 (Apr. 1982), pp. 29–32. URL: https://doi.org/10.1145/1005937.1005943.

[176] Carter McNamara. *General Guidelines for Conducting Research Interviews*. May 2010. URL: https://managementhelp.org/businessresearch/interviews.htm (visited on 04/24/2019).

[177] Dominic Merten. *Usability-Evaluierung ausgewählter Scrum-Tools samt konzeptioneller Verbesserungsvorschläge für scrummage*. Master Thesis. Paderborn University, Sept. 2016.

[178] Microsoft. *Add portfolio backlogs - Azure DevOps & TFS*. Dec. 2017. URL: https://docs.microsoft.com/en-us/azure/devops/reference/add-portfolio-backlogs (visited on 03/23/2020).

[179] D. Millington and J. Stapleton. "Developing a RAD standard." In: *IEEE Software* 12.5 (Sept. 1995), pp. 54–55. URL: https://doi.org/10.1109/52.406757.

[180] Margaret R. Minsky. "Manipulating simulated objects with real-world gestures using a force and position sensitive screen." In: *ACM SIGGRAPH Computer Graphics* 18.3 (Jan. 1984), pp. 195–203. URL: https://doi.org/10.1145/964965.808598.

[181] Nils Brede Moe and Torgeir Dingsøyr. "Scrum and Team Effectiveness: Theory and Practice." In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Pekka Abrahamsson et al. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2008, pp. 11–20. URL: https://doi.org/10.1007/978-3-540-68255-4_2.

[182] Mozilla Corporation. *The WebSocket API (WebSockets) - Web APIs | MDN*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (visited on 08/17/2022).

[183] Jesse Mu and Advait Sarkar. "Do We Need Natural Language? Exploring Restricted Language Interfaces for Complex Domains." In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI EA '19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–6. URL: https://doi.org/10.1145/3290607.3312975.

[184] C. Murad et al. "Revolution or Evolution? Speech Interaction and HCI Design Guidelines." In: *IEEE Pervasive Computing* 18.2 (Apr. 2019), pp. 33–45. URL: https://doi.org/10.1109/MPRV.2019.2906991.

[185] H. Neukom. "The Second Life of ENIAC." In: *IEEE Annals of the History of Computing* 28.2 (Apr. 2006), pp. 4–16. URL: https://doi.org/10.1109/MAHC.2006.39.

[186] J. von Neumann. "First draft of a report on the EDVAC." In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. URL: https:/doi.org/10.1109/85.238389.

[187] Jakob Nielsen. "Finding usability problems through heuristic evaluation." In: *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '92*. Monterey, California, United States: ACM Press, 1992, pp. 373–380. URL: https://doi.org/10.1145/142750.142834.

[188] Jakob Nielsen. "Enhancing the Explanatory Power of Usability Heuristics." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. New York, NY, USA: ACM, 1994, pp. 152–158. URL: https://doi.org/10.1145/191666.191729.

[189] Jakob Nielsen. *How to Conduct a Heuristic Evaluation*. Nov. 1994. URL: https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/ (visited on 02/24/2019).

[190] Jakob Nielsen. *Mouse vs. Fingers as Input Device*. Apr. 2012. URL: https://www.nngroup.com/articles/mouse-vs-fingers-input-device/ (visited on 01/13/2022).

[191] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. Oct. 2020. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (visited on 06/04/2023).

[192] Jakob Nielsen and Thomas K. Landauer. "A Mathematical Model of the Finding of Usability Problems." In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. New York, NY, USA: ACM, 1993, pp. 206–213. URL: https://doi.org/10.1145/169059.169166.

[193] Jakob Nielsen and Rolf Molich. "Heuristic Evaluation of User Interfaces." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '90. New York, NY, USA: ACM, 1990, pp. 249–256. URL: https://doi.org/10.1145/97243.97281.

[194] Erik Nilsson. "Design Patterns for User Interface for Mobile Applications." In: *Advances in Engineering Software* 40 (Jan. 2009), pp. 1318–1328. URL: https://doi.org/10.1007/978-1-84882-206-1_28.

[195] Don Norman. *The Invisible Computer*. The MIT Press, 1999.

[196] Donald A. Norman and Jakob Nielsen. "Gestural interfaces: a step backward in usability." In: *Interactions* 17.5 (Sept. 2010), pp. 46–49. URL: https://doi.org/10.1145/1836216.1836228.

[197] Nuigroup. *Wiki: Natural User Interface*. 2014. URL: http://wiki.nuigroup.com/Natural_User_Interface#Publications (visited on 10/10/2014).

[198] Vyacheslav Olshevsky et al. "Touchless Gestures for Interactive Messaging: Gesture Interface for Sending Emoji." In: *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*. MobileHCI '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1–4. URL: https://doi.org/10.1145/3406324.3410535.

[199] Barry Overeem. *Daily Scrum - Tips & Tactics*. Sept. 2015. URL: https://www.scrum.org/resources/blog/daily-scrum-tips-tactics (visited on 05/04/2019).

[200] Barry Overeem. *Jira - A Necessary Evil?* Dec. 2016. URL: https://www.scrum.org/resources/blog/jira-necessary-evil (visited on 03/18/2020).

[201] K. V. Jeeva Padmini et al. "Challenges Faced by Agile Testers: A Case Study." In: *2018 Moratuwa Engineering Research Conference (MERCon)*. May 2018, pp. 431–436. URL: https://doi.org/10.1109/MERCon.2018.8421968.

[202] Stephen R. Palmer. *Practical Guide to Feature-Driven Development, A*. Upper Saddle River, NJ: Prentice Hall, Feb. 2002.

[203] Shelly Park et al. "An interactive speech interface for summarizing agile project planning meetings." In: *CHI '06 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '06. New York, NY, USA: Association for Computing Machinery, Apr. 2006, pp. 1205–1210. URL: https://doi.org/10.1145/1125451.1125677.

[204] D. L. Parnas and P. C. Clements. "A rational design process: How and why to fake it." In: *IEEE Transactions on Software Engineering* SE-12.2 (Feb. 1986), pp. 251–257. URL: https://doi.org/10.1109/TSE.1986.6312940.

[205] Jeff Patton. *Don't Know What I Want, But I Know How to Get It*. Jan. 2008. URL: https://www.jpattonassociates.com/dont_know_what_i_want/ (visited on 10/01/2021).

[206] Jeff Patton. *User Story Mapping: Discover the Whole Story, Build the Right Product*. 1st ed. Beijing ; Sebastopol, CA: O'Reilly and Associates, Oct. 2014.

[207] Esben Warming Pedersen and Kasper Hornbæk. "An experimental comparison of touch interaction on vertical and horizontal surfaces." In: *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*. NordiCHI '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 370–379. URL: https://doi.org/10.1145/2399016.2399074.

[208] Roman Pichler. *Agile Product Management with Scrum: Creating Products that Customers Love*. 1st ed. Upper Saddle River, NJ: Addison-Wesley Professional, Mar. 2010.

[209] Roman Pichler. *A Template for Formulating Great Sprint Goals*. Mar. 2014. URL: https://www.romanpichler.com/blog/sprint-goal-template/ (visited on 12/07/2019).

[210] Roman Pichler. *Creating Effective Sprint Goals*. Dec. 2022. URL: https://www.romanpichler.com/blog/effective-sprint-goals/ (visited on 02/03/2024).

[211] H. Polachek. "History of the journal Mathematical Tables and other Aids to Computation, 1959-1965." In: *IEEE Annals of the History of Computing* 17.3 (1995), pp. 67–74. URL: https://doi.org/10.1109/85.397062.

[212] Alexei Quapp. *Konzeption und Weiterentwicklung eines Tools zur Unterstützung des Review-Prozesses in Scrum*. Master Thesis. Paderborn University, Jan. 2017.

[213] L. B. S. Raccoon. "Fifty years of progress in software engineering." In: *ACM SIGSOFT Software Engineering Notes* 22.1 (Jan. 1997), pp. 88–104. URL: https://doi.org/10.1145/251759.251878.

[214] Brian Randell. "The 1968/69 nato software engineering reports." In: *History of Software Engineering* (1996), p. 37.

[215] Fred Reichheld. *The Ultimate Question: Driving Good Profits and True Growth*. 1st ed. Boston, Mass: Harvard Business School Press, Mar. 2006.

[216] Stephan Richter, Christian Holz, and Patrick Baudisch. "Bootstrapper: recognizing tabletop users by their shoes." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 1249–1252. URL: https://doi.org/10.1145/2207676.2208577.

[217] Enrique TOPO Rodríguez. *Rugby: The Art of Scrummaging: A History, a Manual and a Law Dissertation on the Rugby Scrum*. 1st reprint 2018 of 1st edition 2014. Meyer & Meyer Sport, Apr. 2018.

[218] Yvonne Rogers and Siân Lindley. "Collaborating Around Vertical and Horizontal Large Interactive Displays: Which Way Is Best?" In: *Interacting with Computers* 16 (Dec. 2004), pp. 1133–1152. URL: https://doi.org/10.1016/j.intcom.2004.07.008.

[219] Yvonne Rogers, Helen Sharp, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. 4th ed. Chichester: Wiley John + Sons, May 2015.

[220] Raúl Rojas and Ulf Hashagen. *The First Computers History and Architectures*. Cambridge, MA, USA: MIT Press, 2000.

[221] Ron Jeffries. *What is Extreme Programming?* Mar. 2011. URL: https://ronjeffries.com/xprog/what-is-extreme-programming/ (visited on 01/04/2024).

[222] Mario Rose. *Tool-Support zur Qualitätssicherung in agilen Scrum Projekten*. Master Thesis. Paderborn University, 2016.

[223] Jörg Roth and Claus Unger. "Using Handheld Devices in Synchronous Collaborative Scenarios." In: *Handheld and Ubiquitous Computing*. Ed. by Peter Thomas and Hans-W. Gellersen. Berlin, Heidelberg: Springer, 2000, pp. 187–199. URL: https://doi.org/10.1007/3-540-39959-3_14.

[224] Dr Winston W Royce. "Managing the development of large software systems." In: *Proceedings IEEE WESCON*. 1970, pp. 328–338.

[225] Jessica Rubart. "A Cooperative Multitouch Scrum Task Board for Synchronous Face-to-Face Collaboration." In: *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces - ITS '14*. Dresden, Germany: ACM Press, 2014, pp. 387–392. URL: https://doi.org/10.1145/2669485.2669551.

[226] K.S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Signature Series (Cohn). Pearson Education, 2013.

[227] Nayan B. Ruparelia. "Software development lifecycle models." In: *ACM SIGSOFT Software Engineering Notes* 35.3 (May 2010), p. 8. URL: https://doi.org/10.1145/1764810.1764814.

[228] K. Ryall et al. "Experiences with and observations of direct-touch tabletops." In: *First IEEE international workshop on horizontal interactive human-computer systems (TABLETOP '06)*. 2006, 8 pp.–. URL: https://doi.org/10.1109/TABLETOP.2006.12.

[229] Lawrence J Sanna et al. "The hourglass is half full or half empty: Temporal framing and the group planning fallacy." In: *Group Dynamics: Theory, Research, and Practice* 9.3 (2005), p. 173. URL: https://doi.org/10.1037/1089-2699.9.3.173.

[230] Scaled Agile Inc. *SAFe 6.0 Framework*. 2024. URL: https://scaledagileframework.com/ (visited on 01/14/2024).

[231] Gianluca Schiavo et al. "Evaluating an automatic rotation feature in collaborative tabletop workspaces." In: *CHI '11 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 1315–1320. URL: https://doi.org/10.1145/1979742.1979767.

[232] Sebastian Schmidt et al. "A set of multi-touch graph interaction techniques." In: *ACM International Conference on Interactive Tabletops and Surfaces - ITS '10*. Saarbrücken, Germany: ACM Press, 2010, p. 113. URL: https://doi.org/10.1145/1936652.1936673.

[233] Ulrike Schultze and Michel Avital. "Designing interviews to generate rich data for information systems research." In: *Information and Organization* 21.1 (Jan. 2011), pp. 1–16. URL: https://doi.org/10.1016/j.infoandorg.2010.11.001.

[234] Ken Schwaber. "SCRUM Development Process." In: *Business Object Design and Implementation*. Ed. by Jeff Sutherland et al. London: Springer London, 1997, pp. 117–134. URL: https://doi.org/10.1007/978-1-4471-0947-1_11.

[235] Ken Schwaber. *About*. 2020. URL: https://www.scrum.org/about (visited on 11/28/2020).

[236] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[237] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. Nov. 2017. URL: https://scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf (visited on 11/19/2018).

[238] Jan Schwarzer et al. "Ambient Surfaces: Interactive Displays in the Informative Workspace of Co-located Scrum Teams." In: *Proceedings of the 9th Nordic Conference on Human-Computer Interaction*. NordiCHI '16. Gothenburg, Sweden: Association for Computing Machinery, Oct. 2016, pp. 1–4. URL: https://doi.org/10.1145/2971485.2971493.

[239] Johannes Schöning et al. "Building Interactive Multi-Touch Surfaces." In: *J. Graphics, GPU, & Game Tools* 14 (Jan. 2009), pp. 35–55. URL: https://doi.org/10.1080/2151237X.2009.10129285.

[240] Stacey D. Scott, M. Sheelagh T. Carpendale, and Kori Inkpen. "Territoriality in collaborative tabletop workspaces." In: *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. CSCW '04. New York, NY, USA: Association for Computing Machinery, Nov. 2004, pp. 294–303. URL: https://doi.org/10.1145/1031607.1031655.

[241] Stacey D. Scott, Karen D. Grant, and Regan L. Mandryk. "System Guidelines for Co-located, Collaborative Work on a Tabletop Display." In: *ECSCW 2003*. Ed. by Kari Kuutti et al. Dordrecht: Springer Netherlands, 2003, pp. 159–178. URL: https://doi.org/10.1007/978-94-010-0068-0_9.

[242] Scrum Inc. *The Official Scrum@Scale Guide*. 2021. URL: https://www.scrumatscale.com/scrum-at-scale-guide/ (visited on 01/14/2024).

[243] Scrum.org. *2019 Scrum Master Trends Report*. 2019. URL: https://www.scrum.org/resources/2019-scrum-master-trends-report (visited on 11/28/2020).

[244] Scrum.org. *Professional Scrum Certified Count*. 2020. URL: https://www.scrum.org/professional-scrum-certifications/count (visited on 11/28/2020).

[245] Scrum.org. *Professional Scrum Master™ III*. 2020. URL: https://www.scrum.org/professional-scrum-master-iii-certification (visited on 12/01/2020).

[246] Scrum.org. *What is ScrumBut?* 2020. URL: https://www.scrum.org/resources/what-scrumbut (visited on 12/13/2020).

[247] Scrum.org. *What is a Daily Scrum?* 2020. URL: https://www.scrum.org/resources/what-is-a-daily-scrum (visited on 03/08/2020).

[248] Scrum.org. *Online Nexus Guide | Scrum.org*. Jan. 2021. URL: https://www.scrum.org/resources/online-nexus-guide (visited on 01/14/2024).

[249] T. Sedano, P. Ralph, and C. Péraire. "The Product Backlog." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 200–211. URL: https://doi.org/10.1109/ICSE.2019.00036.

[250] Dorothy Shamonsky. *The Idea of a Natural User Interface is Not Naturally Easy to Grasp*. June 2019. URL: https://medium.com/swlh/the-idea-of-a-natural-user-interface-is-not-naturally-easy-to-grasp-a4a5a9160be9 (visited on 03/11/2021).

[251] Vibhu Saujanya Sharma and Vikrant Kaulgud. "Agile Workbench: Tying People, Process, and Tools in Distributed Agile Delivery." In: *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*. Aug. 2016, pp. 69–73. URL: https://doi.org/10.1109/ICGSE.2016.17.

[252] Ben Shneiderman et al. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 6th ed. Boston: Pearson, Apr. 2016.

[253] Garth Shoemaker et al. "Mid-air text input techniques for very large wall displays." In: *Proceedings of Graphics Interface 2009*. GI '09. CAN: Canadian Information Processing Society, May 2009, pp. 231–238. URL: https://dl.acm.org/doi/10.5555/1555880.1555931.

[254] David Skinner. "The Age of Female Computers." In: *The New Atlantis - a journal of technology & society* Spring 2006 (2006), pp. 96–103. URL: https://www.thenewatlantis.com/publications/the-age-of-female-computers.

[255] James P. Spradley. *The Ethnographic Interview*. New York: Thomson Learning, Apr. 1979.

[256] Jennifer Stapleton. *DSDM, Dynamic Systems Development Method: The Method in Practice*. Cambridge University Press, 1997.

[257] Martin Stember. *Konzeption und prototypische Entwicklung eines Taskboards als Tangible User Interface zur Unterstützung des agilen Vorgehensmodells Scrum*. Master Thesis. Paderborn University, 2018.

[258] Steve Messenger. *Chapter 6: Process*. 2014. URL: https://www.agilebusiness.org/page/ProjectFramework_06_Process (visited on 07/30/2019).

[259] Jeff Sutherland. *Story Points: Why are they better than hours?* May 2013. URL: https://www.scruminc.com/story-points-why-are-they-better-than/ (visited on 12/31/2019).

[260] Jeff Sutherland and J. J. Sutherland. *Scrum: The Art of Doing Twice the Work in Half the Time*. 1st edition. New York: Currency, Sept. 2014.

[261] Mohsen Taheri and S. Masoud Sadjadi. "A Feature-Based Tool-Selection Classification for Agile Software Development." In: July 2015, pp. 700–704. URL: https://doi.org/10.18293/SEKE2015-234.

[262] Hirotaka Takeuchi and Ikujiro Nonaka. "The New New Product Development Game." In: *Harvard Business Review* 64.1 (1986), p. 12.

[263] Hao Tang et al. "GestureGAN for Hand Gesture-to-Gesture Translation in the Wild." In: *Proceedings of the 26th ACM international conference on Multimedia*. MM '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 774–782. URL: https://doi.org/10.1145/3240508.3240704.

[264] The LeSS Company B.V. *LeSS Framework*. 2024. URL: https://less.works/less/framework (visited on 01/14/2024).

[265] Steven Thomas. *Revisiting the Iterative Incremental Mona Lisa*. Dec. 2012. URL: http://itsadeliverything.com/revisiting-the-iterative-incremental-mona-lisa (visited on 01/03/2019).

[266] Chad Tossell et al. "Characterizing web use on smartphones." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 2769–2778. URL: https://doi.org/10.1145/2207676.2208676.

[267] Daniel W Turner. "Qualitative Interview Design: A Practical Guide for Novice Investigators." In: *The Qualitative Report* 15.3 (2010), pp. 754–760. URL: https://nsuworks.nova.edu/tqr/vol15/iss3/19.

[268] Maria Eloina Pelaez Valdez. "A Gift From Pandora's Box - The Software Crisis." PhD thesis. Edinburgh: University of Edinburgh, 1988.

[269] Veli-Pekka Eloranta, Kai Koskimies, and Tommi Mikkonen. "Exploring ScrumBut—An empirical study of Scrum anti-patterns." In: *Information and Software Technology* 74 (June 2016), pp. 194–203. URL: https://doi.org/10.1016/j.infsof.2015.12.003.

[270] VersionOne. *9th Annual State of Agile Report*. Apr. 2017. URL: https://www.stateofagile.com/#ufh-i-338592759-9th-annual-state-of-agile-report/473508 (visited on 02/12/2019).

[271] VersionOne. *13th Annual State of Agile Survey | The Largest, Longest-Running Agile Survey*. Feb. 2019. URL: https://stateofagile.versionone.com/ (visited on 02/12/2019).

[272] Frank Vetere et al. "Social NUI: social perspectives in natural user interfaces." In: *Proceedings of the 2014 companion publication on Designing interactive systems*. DIS Companion '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 215–218. URL: https://doi.org/10.1145/2598784.2598802.

[273] Dr. Hendrik Vollrath. *START-UP-Hochschul-Ausgründungen NRW*. 2018. URL: https://www.efre.nrw.de/wege-zur-foerderung/weitere-foerderprogramme/start-up-transfernrw/start-up-hochschul-ausgruendungen-nrw/ (visited on 07/30/2023).

[274] Henning Voss and Georg Schneider. "Nori Scrum meeting table." In: *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*. ITS '09. New York, NY, USA: Association for Computing Machinery, Nov. 2009, p. 1. URL: https://doi.org/10.1145/1731903.1731956.

[275] Gerard Wagenaar, Sietse Overbeek, and Remko Helms. "Describing Criteria for Selecting a Scrum Tool Using the Technology Acceptance Model." In: *Intelligent Information and Database Systems*. Ed. by Ngoc Thanh Nguyen et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 811–821. URL: https://doi.org/10.1007/978-3-319-54430-4_77.

[276] Bill Wake. *INVEST in Good Stories, and SMART Tasks*. Aug. 2003. URL: https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/ (visited on 01/26/2019).

[277] Bill Wake. *Independent Stories in the INVEST Model*. Feb. 2012. URL: https://xp123.com/articles/independent-stories-in-the-invest-model/ (visited on 01/26/2019).

[278] Feng Wang and Xiangshi Ren. "Empirical evaluation for finger input properties in multi-touch interaction." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. New York, NY, USA: Association for Computing Machinery, Apr. 2009, pp. 1063–1072. URL: https://doi.org/10.1145/1518701.1518864.

[279] Diane Watson et al. "Deconstructing the touch experience." In: *Proceedings of the 2013 ACM international conference on Interactive tabletops and surfaces*. ITS '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 199–208. URL: https://doi.org/10.1145/2512349.2512819.

[280] Sebastian Weber et al. "APDT: An Agile Planning Tool for Digital Tabletops." In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Pekka Abrahamsson et al. Lecture Notes in Business Information Processing. Springer

Berlin Heidelberg, 2008, pp. 202–203. URL: https://doi.org/10.1007/978-3-540-68255-4_21.

[281] M. Weiser, R. Gold, and J. S. Brown. "The origins of ubiquitous computing research at PARC in the late 1980s." In: *IBM Systems Journal* 38.4 (1999), pp. 693–696. URL: https://doi.org/10.1147/sj.384.0693.

[282] Mark Weiser. "The Computer for the 21 st Century." In: *Scientific American* 265.3 (1991), pp. 94–105. URL: http://www.jstor.org/stable/24938718.

[283] Mark Weiser and John Seely Brown. "Designing calm technology." In: *Xerox PARC* (1995). URL: https://calmtech.com/papers/designing-calm-technology (visited on 10/15/2021).

[284] Pierre Wellner. "Interacting with paper on the DigitalDesk." In: *Communications of the ACM* 36.7 (July 1993), pp. 87–96. URL: https://doi.org/10.1145/159544.159630.

[285] H. G. Wells. *H.G. Wells - The Sea Lady: "Human history in essence is the history of ideas."* Horse's Mouth, Feb. 2017.

[286] Dave West. "Water-Scrum-Fall Is The Reality Of Agile For Most Organizations Today." In: *Application Development & Delivery Professionals* Forrester Research, Inc (2011), p. 17. URL: https://www.verheulconsultants.nl/water-scrum-fall_Forrester.pdf (visited on 10/20/2020).

[287] Matthew T. West. "Ubiquitous computing." In: *Proceedings of the 39th annual ACM SIGUCCS conference on User services*. SIGUCCS '11. New York, NY, USA: Association for Computing Machinery, Nov. 2011, pp. 175–182. URL: https://doi.org/10.1145/2070364.2070410.

[288] Jerome White and Mayuri Duggirala. "Speech-interface prompt design: lessons from the field." In: *Proceedings of the Seventh International Conference on Information and Communication Technologies and Development*. ICTD '15. New York, NY, USA: Association for Computing Machinery, May 2015, pp. 1–4. URL: https://doi.org/10.1145/2737856.2737861.

[289] Michael R. Williams. *A History of Computing Technology, 2nd Edition*. 2 edition. Los Alamitos, Calif: Wiley-IEEE Computer Society Pr, Mar. 1997.

[290] Niklaus Wirth. "A Brief History of Software Engineering." In: *IEEE Annals of the History of Computing* 30.3 (July 2008), pp. 32–39. URL: https://doi.org/10.1109/MAHC.2008.33.

[291] Stefan Wolpers. *Sprint Review Anti-Patterns: 15 Ways how Scrum Teams Can Improve*. Nov. 2019. URL: https://age-of-product.com/sprint-review-anti-patterns/ (visited on 03/08/2020).

[292] Stefan Wolpers. *The Daily Scrum: 16 Stand-up Anti-Patterns to Avoid*. Oct. 2019. URL: https://age-of-product.com/stand-up-anti-patterns/ (visited on 03/07/2020).

[293] Stefan Wolpers. *Scrum: 20 Sprint Planning Anti-Patterns*. Jan. 2020. URL: https://age-of-product.com/scrum-sprint-planning-anti-patterns/ (visited on 03/04/2020).

[294] Xin Wang and Frank Maurer. "Tabletop AgilePlanner: A tabletop-based project planning tool for agile software development teams." In: *2008 3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems*. Amsterdam: IEEE, Oct. 2008, pp. 121–128. URL: https://doi.org/10.1109/TABLETOP.2008.4660194.

[295] Raymond T. Yeh. "System Development as a Wicked Problem." In: *International Journal of Software Engineering and Knowledge Engineering* 1.2 (1991), pp. 117–130. URL: https://doi.org/10.1142/S0218194091000123.

[296] Gayle Young. *HughLeCaine.com - Instruments*. 1999. URL: http://www.hughlecaine.com/en/instruments.html (visited on 03/15/2021).

[297] Ulrich von Zadow et al. "YouTouch! Low-Cost User Identification at an Interactive Display Wall." In: *Proceedings of the International Working Conference on Advanced Visual Interfaces*. AVI '16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 144–151. URL: https://doi.org/10.1145/2909132.2909258.

[298] Hind Zahraoui and Mohammed Abdou Janati Idrissi. "Adjusting story points calculation in scrum effort time estimation." In: *2015 10th International Conference on Intelligent Systems: Theories and Applications (SITA)*. Oct. 2015, pp. 1–8. URL: https://doi.org/10.1109/SITA.2015.7358400.

[299] digital.ai. *15th Annual State Of Agile Report*. July 2021. URL: https://stateofagile.com/ (visited on 01/13/2024).

[300] digital.ai. *16th Annual State Of Agile Report*. July 2022. URL: https://stateofagile.com/ (visited on 01/13/2024).

[301] t2informatik. *What is an Use Case?* Oct. 2019. URL: https://t2informatik.de/en/smartpedia/use-case/ (visited on 01/21/2024).

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AI	Artificial Intelligence
ALM	Application Lifecycle Management
APDT	Agile Planner for Digital Tabletops
API	Application Programming Interface
BRL	Ballistic Research Laboratory
CAD	Computer Aided Design
CASE	Computer Aided Software Engineering
CLI	Command Line Interfaces
CRUD	Create Read Update Delete
DOD	Definition of Done
DOF	Degrees of freedom
DOR	Definition of Ready
DSDM	Dynamic Systems Development Method
DSI	Diffused Surface Illumination
EDVAC	Electronic Discrete Variable Automatic Computer
ENIAC	Electronic Numerical Integrator And Computer
FDD	Feature-Driven Development
FTIR	Frustrated Total Internal Reflection
GUI	Graphical User Interface
HCI	Human-Computer Interaction
IDE	Integrated Development Environment
IR	Infrared
IID	Iterative and Incremental Development
JAD	Joined Application Design
LLP	Laser Light Plane
MAM	Marble Answering Machine
MVP	Minimum Viable Product
NACA	National Advisory Committee for Aeronautics
NASA	National Air and Space Administration
NPS	Net Promoter Score
NUI	Natural User Interface
OKR	Objectives Key Result
PARC	Palo Alto Research Center
PC	Personal Computer
PBI	Product Backlog Item
PSM	Professional Scrum Master
RAD	Rapid Application Development
RDI	Rear Diffused Illumination
RIPP	Rapid Iterative Production Prototyping
RQ	Research Question
RUP	Rational Unified Process

SDLC	Software Development Lifecycle
SPA	Single Page Application
SWAT	Specialists With Advanced Tools
TUI	Tangible User Interface
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UX	User Experience
VR	Virtual Reality
VUI	Voice User Interface
WIMP	Windows, Icons, Menus, Pointers
WPA	Works Project Administration
XP	Extreme Programming