Documentation in Continuous Software Development

Theo Theunissen

SIKS Dissertation Series No. XXXX-XX
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Cover illustration: Judith van Beukering
ISBN
NUGI 980
All rights reserved.
Copyright © 2023 Theo Theunissen

Documentation in Continuous Software Development

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

te verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof.dr. H.R.B.M. Kummeling,
ingevolge het besluit van het college voor promoties
in het openbaar te verdedigen op

DAYNAME DD MONTHNAME des middags te TIME uur

door

Theo Theunissen
geboren op 2 november 1964,
te Overasselt

promotor:
prof.dr. S. Brinkkemper

co-promotoren:
dr. S.J.B.A. Hoppenbrouwers
dr. S.J. Overbeek

This thesis would not have been possible without the support and guidance provided by HAN University of Applied Sciences, Arnhem.

Contents

Acronyms vii

1 Introduction 9

2 The Disappearance of Technical Specifications in Web and Mobile Applications 15

3 Specification in Continuous Software Development 23

4 Software Specification and Documentation in Continuous Software Development - A Focus Group Report 39

5 A Mapping Study on Documentation in Continuous Software Development 51

6 Tools are the Message for Communication 83

7 Continuous Learning with the Sandwich of Happiness and Result Planning 97

8 Approaches for Documentation in Continuous Software Development 107

9 Evaluation of Approaches for Documentation in Continuous Software Development 127

10 Conclusion 139

Bibliography 145

Summaries 167

Curriculum Vitae 169

SIKS Dissertations 171

Acronyms

ADR Architecture Decision Record.

ANN Artificial Neural Network.

API Application Programming Interface (API).

ATDD Acceptance Test Driven Development.

BDD Behavior Driven Development.

BoW Bag of Words.

CI Continuous Integration.

CI/CD Continuous Integration/Continuous Deployment.

CNN Convolutional Neural Network.

CORBA Common Object Request Broker Architecture.

CSD Continuous Software Development.

DCAR Decision Centric Architecture Review.

DDL Data Definition Language.

DML Data Manipulation Language.

DoD Definition of Done.

DS Design Science.

ED Executable Documentation.

ERD Entity Relationship Diagram.

FTE Full-time Equivalent.

IDE Integrated Development Environment.

IDF Inverse Document Frequency.

IDL Interface Definition Language.

JSON JavaScript Object Notification.

KPI Key Performance Indicator.

LAMP Linux, Apache, MySQL, PHP. A popular development stack.

LLM Large Language Model.

MEAN Mongo, Express, Angular, Nodejs. A popular development stack.

MLR Multivocal Literature Review.

MOOC Masssive Open Online Course.

MVP Minimum Viable Product.

NLP Natural Language Processing.

ORM Object Relational Mapping.

PoC Proof of Concept.

QGS Quasi-Gold Standard.

REST Representational State Transfer.

RNN Recurrent Neural Network.

RP Result Planning.

RUP RUP.

SAD Software Architecture Description.

SAFe Scaled Agile Framework.

SCM Source Control Management.

SCQA Situation, Complication, Question, Answer (SCQA).

SDD Software Design Description.

SLR Systematic Literature Review.

SMART Specific, Measurable, Acceptable, Relevant, Time-bound.

SMS Systematic Mapping Study.

SOAP Simple Object Access Protocol.

SoH Sandwich of Happiness (SoH).

SRS Software Requirements Specification.

SSOT Single Source of Truth (SSOT).

SWE Software Engineering.

TDD Test Driven Development.

TF Term Frequency.

TFIDF Term Frequency Inverse Document Frequency.

TL;DR Too Long; Didn't Read.

TRL Technology Readiness Level.

TTM Time-to-Market.

UI User Interface.

UML Unified Modelling Language.

XP eXtreme Programming.

YAML Yet Another Markup Language.

Chapter 1

Introduction

With the introduction of the Agile Manifesto, Lean software development, and DevOps, software product documentation has been reduced in quantity, resulting in fewer documents, and less quality originating from loose and informal communication. Causes for this, and the knowledge evaporation it entails, are that the Agile Manifesto values working software over comprehensive documentation [1], that Lean software development advocates efficiency, implying everything that does not contribute to customer value should be considered as waste [2], and that DevOps uses infrastructure-as-code for complying to fast Time-to-Market including continuously changing legislation [3].

The term 'documentation' stems from the Latin word *docere* meaning 'teaching' or 'instructing' [4]. We use the term for diverse types of information, including written, visual and verbal artifacts, that transfer knowledge between stakeholders in software development. In this thesis, we distinguish 'information' from 'documentation'. With 'information', we refer to anything that makes a semantic difference [5] as in binary zeros and ones and causes an effect [6].

This section has the following outline: it starts with a historical overview. Next, a conceptual overview is presented, followed by a description of the actual problem. The following section concerns the study design, stating the main research objectives and questions. The study design also includes a method section with a discussion of relevant research frameworks and research methods. Finally, a reading guide pertains to the previously published studies on which this thesis is based.

1.1 Historical Overview

We start the historical overview in the 1960s with the industrialization of software development [7], [8]. The history is categorized into documentation artifacts that we present in three phases: upfront, while building, and after delivery. This overview covers viewpoints from the 1960s up to the year 2000. Subsequently, a literature review was conducted to investigate in detail publications from 2000 onward.

At the start of an iteration or product development cycle, developers may often consider documentation a burden, as stated by Naur in his seminal paper on software engineering [7]. However, Naur also recognized the importance of interface documentation between subsystems to ensure that different software system components work together seamlessly. He also argued that source code could serve as self-explanatory documentation, which can help developers better understand the functionality of the code they are working on [7, p. 32]. Despite these benefits, the information developers' need is often scattered across different sources, such as documentation, code comments, and online forums [9], making it challenging to find relevant information. Furthermore, the distribution of documentation in different devices can pose a challenge for developers who need to access it [7]. As such, development teams should establish transparent documentation practices and ensure that information is readily accessible to all team members. While building software in iterations has become a popular development approach, it has its drawbacks. One issue is that the focus on building small, incremental changes can sometimes distract from the bigger picture of the project's development [9]. This can lead to a lack of cohesion between software components and overall progress. It is also crucial to keep documentation up-to-date throughout the development process to ensure all team members are on the same page and avoid potential misunderstandings [9]. Unfortunately, documentation can fall by the wayside in some cases as teams prioritize development over documenting their progress [7]. Both of these challenges require careful

consideration and planning to avoid negative consequences [7], [9]. Afterward, on delivery, documentation becomes an essential part of any software project. It serves multiple purposes, such as preserving knowledge for end-users with a manual and aiding maintainers and future developers with a description of design decisions. As highlighted by Overton [9], focusing solely on documentation can sometimes distract from the actual development process. However, it is crucial to balance documentation and development to ensure that a software product remains maintainable and understandable [9]. End-user documentation should be included in software products, while technical documentation should be created for future developers and maintainers. Documentation of a software system should also include design decisions, requirements, specifications, and connections of system components, as these are assumptions about how these components communicate together [8].

Furthermore, independent of the mentioned phases, documentation-related aspects that affected documentation drew attention in the 80s and 90s, such as scientific approaches to software engineering [10], [11], software architecture design [12], [13], the management of software development [14], [15], documentation errors [16], and legal requirements for documentation [17], [18].

We found in our studies that, in addition to the issues mentioned, loose, informal ways of communicating, such as verbal communication and whiteboard sketches, are usually not written down or saved systematically. In addition to Overton [9], the software is essential for most businesses, either supportive, at the core, as an enabler or as an accelerator. This introduces dynamics, including cultural shifts, organizational challenges, values, and knowledge-intensive standards. A part of these changes is the extensive use of tools in a widely spread software development ecosystem. Information about software products is scattered throughout these tools in such an ecosystem [19].

1.2 Conceptual Overview

Artifacts that contribute to understanding and improving the documentation are mentioned below. The artifacts are found in the literature review [20], field research [21], and constructing approaches [19]. Further on in this study, artifacts are added for completeness' sake and for the purpose of illustration.

Before starting an iteration or project, it is essential to invest time and effort into preparing several artifacts upfront [19], [20]. These include written documents with diagrams for stakeholder concerns, constraints, risks, context, requirements, and specifications. A typical artifact that covers these types of information is a presentation-like document that includes drawings and text. A codified interface description must also be created to facilitate communication between subsystems. A plan of approach is also crucial, outlining objectives, deliverables, and a timeline for the project's completion [19]. Specifications for Test Driven Development (TDD) and Behavior Driven Development (BDD) are included to ensure quality control throughout the project's life cycle. Finally, Git commits and pull requests should be used to document design decisions for Natural Language Processing (NLP) projects, ensuring transparency and data integrity. By prioritizing these artifacts upfront, project managers and development teams ensure that everyone is aligned with project objectives and can work together efficiently to deliver a high-quality product [19]. Efficiency refers to the combination of 1) working software that adds value to the customer and 2) knowledge capturing and knowledge transfer to build up, use, operate, and maintain afterward. To build software in iterations effectively, it is essential to update documentation, including diagrams, for analysis, design, implementation, testing, and maintenance [19]. Similarly, updating codified interface descriptions for communication between subsystems is essential. These updates ensure consistency and effectiveness in communication throughout the development process, leading to a successful software system that meets stakeholder requirements. Afterward, on delivery, several key elements must be considered to ensure a software project's success and sustainability. These include treating the source-code as the ultimate truth, using annotations as an integral part of the source-code to aid in understanding and maintenance, employing infrastructure-as-code to enable fast Time-to-Market (TTM) in Continuous Integration/Continuous Deployment (CI/CD) workflows, and using Git commits and pull requests to document and communicate design decisions. By prioritizing and incorporating these factors into the development process, teams can create more reliable and maintainable software to better adapt to changing business needs and market conditions.

The ideal situation for preventing knowledge evaporation requires documentation about different types of information, such as whiteboard sketches, strict interface description and captured verbal communication, for a range of users who rely on information about the software product, such as system engineers, maintainers, and end-users.

1.3 An Actual Problem in Documentation

As presented in the previous overviews, software documentation has a long history that is still perceivable in modern software development. Added to the reasons mentioned beforehand, a few other reasons make documentation a difficult problem. These other reasons are:

1. Epistemological issues questioning the nature of knowledge and the relation between mind and reality [22].
2. Cultural shift with respect to deminished attention span [23].
3. Brain developments for digital natives who grew up primarily with screens for processing information instead of paper [24].

We do not set out to solve epistemological issues here, initiate a cultural shift, or do neural brain research. However, some links can be identified that relate to the approaches and artifacts we consider. For epistemological issues in software development, the relationship between mind and reality concerns the relationship between the stakeholder and the software product. In a platonic view[1], the software product refers to the concept, and documentation artifacts refer to the earthly shadows that represent the software product. The documentation artifacts that represent the software product do not identify it but, at best, describe it.

The cultural shift we refer to is the lack of attention span [26] demonstrated by users who want instant satisfaction, as exploited by some websites[2]. Too Long; Didn't Read (TL;DR) is a popular expression on social media and the web that replaces lengthy texts. A quick search[3] on google rendered approximately 13.5 million hits on reddit.com, 150.000 on medium.com, 19.000 on hackernews, and 5.000 on google scholar. Explanations in literature [26], [27] for this short attention span concern a cultural shift and behavioral developments related to the brain processing visual and textual information.

Another recipe for developing a short attention span is the habit of readers of, for example, web pages to scan quickly for highlighted keywords [28]. Furthermore, learners from 'generation Z' are not so much readers of traditional text documents but consumers of information from diagrams, podcasts, short knowledge clips such as videos, and hands-on experiences such as clicking hyperlinks [26], [27].

1.4 Study Design

1.4.1 Objectives and Research Questions

The *objective* of this study is defined in the main research question:

> *What are the necessary and sufficient conditions for effective communication with just enough documentation in Continuous Software Development (CSD), obtaining insight and control to start building, delivering, maintaining, and continuously using a software product?*

'Necessary conditions' refers to the minimal requirements for an event to occur. 'Sufficient conditions' make the event occur. A necessary condition alone is not sufficient. A simple example can make this clear. The necessary conditions for fire are 'air', 'fuel' and 'heat'. However, these necessary conditions become sufficient for fire only when air, fuel, and heat are in a specific configuration. A simple example of a sufficient condition without necessity is 'you traveled to Amsterdam by plane'; the plane is sufficient but unnecessary. *Insight* refers to knowledge and facts that someone knows. *Control* refers to the ability to change the course of events in specific directions. Insight is a necessary condition for control. *Effective communication* refers to results that could not have been achieved without sufficient information. *Documentation in CSD* refers to the domain (scope) for this research. Making the necessary and sufficient explicit helps in understanding which elements are required for concepts like 'insight', 'control', 'effective communication', and 'documentation in CSD', including the relations between these concepts. Additionally, the conditions explicitly clarify how, why, and when the concepts occur.

Underlying the main research question, there are three more manageable research questions.

RQ1 Why is knowledge acquisition, building, preserving, and revealing software development a complex and challenging problem, leading to lesser documentation with lower quality?

[1] According to Whitehead, "The safest general characterization of the European philosophical tradition is that it consists of a series of footnotes to Plato." [25, p. 39]. This makes it safe to refer to ancient history from Plato, 2500 years ago.

[2] See for instance the TikTok recommendation engine and its relation with dopamine [27].

[3] Google this query ' "tl;dr" reddit.com' and replace reddit.com with other domain names.

RQ2 What are defining and contextual characteristics for Continuous Software Development?

RQ3 Which documentation artifacts are required: a) *upfront* for *an individual* to start a project or an iteration, b) *while building* the software product *with team members*, and c) *afterward by others* for deployment, maintenance, and usage (in short: continuation) of a software product?

1.4.2 Research Methods

The utmost thoroughness of methodological evidence requires meta-studies such as secondary or tertiary literature reviews because multiple researchers have reviewed the primary literature with assessment criteria. This type of inter-subjective assessment has the highest degree of objectivity. In the current project, we started with the second in line: a literature review such as a mapping study. Experimental studies have the next lowest level of rigor, including randomized controlled trials, quasi-experimental studies, or cohort studies. Observational studies such as case studies and interviews have the lowest level of rigor, because personal, individual observations are, by definition, subjective and cannot be verified. See Table 1.1a for research methods in each study presented in this Ph.D. thesis.

Rigor	Research Method ↓ / Chapter →	2	3	4	5	6	7	8	9
1	Systematic Mapping Study				✓	✓	✓		✓
1	Multi-vocal Literature Study						✓	✓	✓
2	Document Analysis		✓	✓		✓	✓	✓	✓
2	Survey, Questionnaire	✓	✓	✓	✓		✓	✓	✓
2	Design Science (framework)							✓	✓
3	Expert focus group			✓				✓	✓
4	Case Study						✓	✓	✓
4	Semi-structured Interviews		✓		✓		✓	✓	✓
4	Theory generating interviews			✓			✓	✓	
4	Validation (Design Science)							✓	
4	Evaluation (Design Science)								✓

(a) Research Methods used in the Chapters.

1.4.3 The Scientific Aspiration

The objective of science is to come to new knowledge and new models with the discovery of a problem and the invention of a solution. Most academic studies are applied studies: medicine or law, for example, are only meaningful when applied to people or (legal) entities. Software development studies are also empirical in nature.

Exercise in Methodology

The aspiration for scientific research is paved with research methods. Wohlin, Runeson, Höst, *et al.* propose guidance on three methods of conducting software engineering [10]. These methods are theoretical, experimental, and empirical. According to the authors, theoretical methods are used to develop new models, theories, and frameworks, including approaches. These methods are used to create a conceptual understanding of a phenomenon or problem. Theoretical methods involve a systematic review of literature, conceptual modeling, and synthesis of existing knowledge. They are used to establish a theoretical foundation for empirical research. We introduced three approaches (Chapter 8) for knowledge acquisition and knowledge distribution that fits within the definition provided by Wohlin, Runeson, Höst, *et al.* Experimental methods, on the other hand, are used to investigate cause-and-effect relationships in software engineering. These methods are used to test hypotheses by manipulating independent variables and observing the effect on dependent variables. Experimental methods involve controlled experiments, quasi-experiments, and case studies. These methods are used to establish causal relationships between variables. Experimental methods apply to Chapters 8 and 9. Empirical methods are used to study software engineering phenomena in natural settings. These methods involve collecting data from real-world situations using observation, surveys, interviews, and other methods. Empirical methods, including case studies, surveys, and field experiments, were employed in Chapters 2 through 7 to collect data. These methods are used to understand the context in which software engineering practices are applied and to identify factors that influence software development outcomes. The methods used in this research project have already

been mentioned in Table 1.1a and will be demonstrated in the referred studies. Typically, these methods support a paradigm to do empirical research for collecting and analyzing data, followed by interpreting results.

Models, Logic and Causal Relations

Abstraction can be defined as leaving out everything that is *not* required to understand a problem. A model adds relations to the abstractions for a certain domain. As such, it describes an abstraction of entities, relations, and attributes of the entities and relations. These relations are either logical or causal. Logical relations are deductive, inductive, or abductive. A particular variant of induction and abduction concerns descriptive and inferential statistical relations. With deductive relations (major to minor premise), no new knowledge is added. With induction (from individual cases to general rules), new knowledge might be introduced but cannot be proven. The closest to causal reasoning is abduction, which starts with conjectures that might be rejected but can never be proven [29].

We use a model to:

1. Individually organize facts and values by filtering, grouping, and order to understand the problem.
2. Individually reason about a problem and candidate solutions.
3. Communicate the problem and candidate solutions with stakeholders and team members.
4. Discuss the problem and candidate solutions.
5. Start building with team members.

A sufficient condition can be identified as a cause.

1.5 How to Read This Dissertation?

Figure 1.1 shows the phases in this research project.

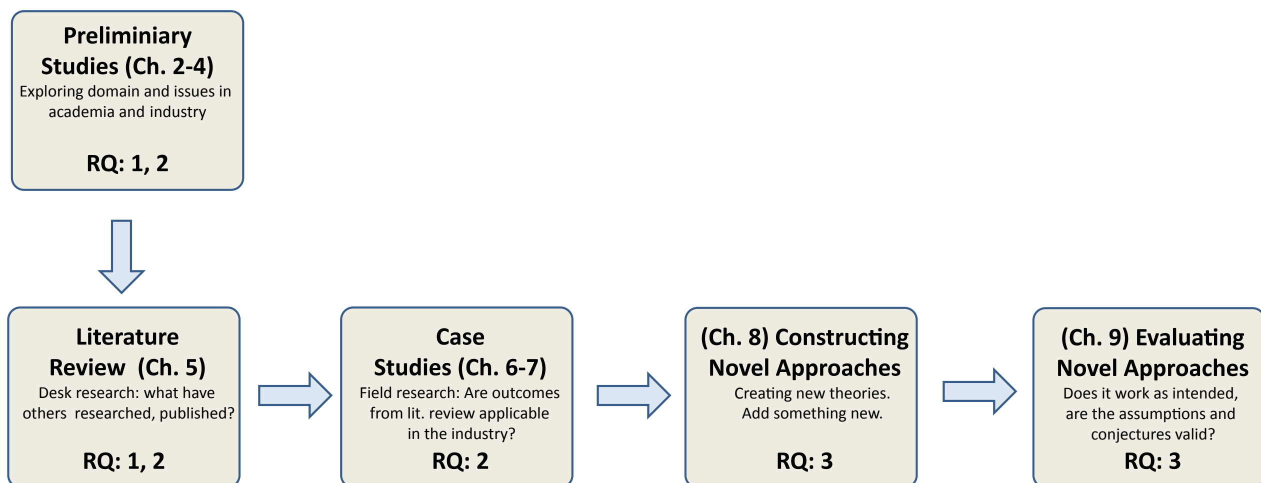


Figure 1.1: Studies in this research project with phases, chapters, and research questions (RQ).

1.5.1 Phase 1, Chapters 2-4, Preliminary Studies

The preliminary studies [30], [S31], [S32] was limited to investigating the domain and collecting issues, problems, and candidate solutions. We did not perform exploratory studies to find appropriate methods. From the start, the phases of exploration, desk research, field research, adding something new, and evaluating novel approaches, were clear. Three studies were conducted to investigate research question 1, which focuses on knowledge acquisition and distribution, and research question 2, which examines CSD.

1.5.2 Phase 2, Chapter 5, Literature Review

In the second phase, a Systematic Mapping Study (SMS) [20] was conducted to obtain an overview of publications on documentation in Continuous Software Development between 2001 and 2019. The result was an overview of 63 studies, of which 40 were related to documentation practices and challenges, and 23 were related

to tools in CSD. We decided to conduct a SMS and not a Systematic Literature Review (SLR). A SMS is typically used when there are few or no secondary studies; the main objective of an SMS is to classify and conduct a thematic analysis of the literature. In Section 2 of the SMS, other reasons are mentioned that validate the performance of this type of literature review. Through this study, a more thorough exploration and definition of both research question 1 and research question 2 were undertaken.

1.5.3 Phase 3, Chapter 6-7, Case Studies

The third phase concerned an overview of practices and challenges for practitioners in the industry [21], [33]. We used multiple units of analysis -15 individual participants- in five multiple case studies, including departments within organizations, for validating and extending the SMS. With this study, we validated the SMS and gained new insights from the industry. In this study, the software development ecosystem was defined, including types of information and the amount of structure of information. By conducting these studies, research question 2 was addressed, and a definitive definition of CSD was established.

1.5.4 Phase 4, Chapter 8, Novel Approaches

In the fourth phase, we presented three approaches, including a total of 33 artifacts [19]. The three approaches are 'Just enough Upfront', 'Executable Documentation', and 'Automatic Text Analysis'. For the approaches, conditions were defined, including characteristics for applicability. Characteristics for the approaches cover the maturity of stakeholder concerns, requirements, technology, processes, and demand for fast TTM distinguish approaches. By introducing this chapter, we lay the groundwork for tackling research question 3, which concerns the types of knowledge necessary before deployment versus during deployment, use, and maintenance.

1.5.5 Phase 5, Chapter 9, Evaluating Approaches

In the last phase of this research project, an evaluation was conducted in industry and education to assess the applicability, hurdles, and requirements for implementing the approaches **theunissen2023evaluationapproachesdocumentat**
Our research investigates the idea that there is a notable difference in documentation practices between open source and closed source. Furthermore, we explore the assumption that there is also another category: closed *classified* source, that adds security demands because of a criminal investigation or national defense. Knowledge preservation is best documented in open-source projects and best kept in hearts and minds in closed classified sources. Examining these approaches, we tackle and complete research question 3 concerning the upfront and subsequent knowledge requirements.

Chapter 2

The Disappearance of Technical Specifications in Web and Mobile Applications[1]

Abstract In recent years, we have been observing a paradigm shift in design and documentation practices for web and mobile applications.There is a trend towards fewer up-front design specification and more code and configuration-centric documentation. In this paper we present the results of a survey, conducted with professional software engineers who build web and mobile applications. Our focus was on understanding the role of software architecture in these applications, i.e. what is designed up-front and how; which parts of the architecture are reused from previous projects and what is the average lifetime of such applications.Among other things, the results indicate that free-text design specification is favored over the use of modeling languages like Unified Modelling Language (UML); architectural knowledge is primarily preserved through verbal communication between team members, and the average lifetime of web and mobile applications is between one and five years.

Keywords Concept Documentation, Documentation Practices, Free Text Documentation, Mobile Application Design, Verbal Communication

2.1 Introduction

In recent years, we have been observing a paradigm shift in the software engineering community. Professional software development projects traditionally relied on upfront planning and design, distinct software phases and often a clear separation of roles and responsibilities within project teams. Ever growing time-to-market constraints, however, leads to high innovation pressure, which brought forth methods and techniques like agile project management, continuous delivery, and DevOps, which break with the traditional way of approaching software projects. Apart from this, primarily for web and mobile application development, developers now need to deal with an increasingly heterogeneous tool and language stack. In that domain in particular, software is often designed ad hoc, or at best by drawing informal sketches on a white board while discussing with peers. The reasons for this are multifold: where applications must be developed and rolled out quickly, designers do not seem to see the value of spending much time on modeling and documenting solutions. A second reason is that software engineering has no guidelines for efficient modeling of heterogeneous multi-paradigm applications. This is partly due to the fact that software engineering curricula at universities are still heavily focused on

[1]This work was originally published as:
T. Theunissen and U. van Heesch, "The Disappearance of Technical Specifications in Web and Mobile Applications," in *Software Architecture*, vol. 9839, B. Tekinerdogan and U. Zdun, Eds. Springer International Publishing, 2016, pp. 265–273. doi: 10.1007/978-3-319-48992-6_20.

traditional object-oriented analysis and design using UML, which is not a good fit for applications that are not purely object-oriented. In this paper, we describe a questionnaire-based survey, conducted to understand current design and documentation practices that companies use for developing web and mobile applications. Our investigation leads to the conclusion that in the design phase, "experimentation" with *proof of concepts* has the highest score, and "free-text documentation" is more used than technical documentation (UML, Entity Relationship Diagram (ERD)). The continuity of knowledge is primarily achieved using free-text documentation and verbal communication.

The study is part of a larger research project, in which we develop an architecture framework that is streamlined for modern web and mobile applications. In the framework, we plan to provide just-enough architecture and design specification for supporting agile web and mobile application development, while preserving core decisions and design for re-use in subsequent projects.

The rest of this paper is organized as follows. Section 2 describes the research questions and presents and motivates the study design. Section 2 presents the results of the questionnaire and our interpretations with respect to the research questions. The next section presents potential threats to validity. Finally, we conclude and present directions for future work.

2.2 Study Design

In this section, we present our research questions and the study design.

2.2.1 Research Questions

As described in the introduction, the goal of our research project is to develop an architecture framework that optimally supports software engineers in building and maintaining web and mobile applications. The study presented was conducted to settle the baseline process and to get a better understanding of current design and documentation processes, as well as developers' concerns in the industry. In particular, we address the following research questions:

RQ1 How are web and mobile applications designed and how is design knowledge preserved in the industry?

RQ2 Which parts of the software architecture are re-used across web and mobile applications within a development team?

RQ3 What is the average life expectancy of web and mobile applications?

The first question aims at finding out how web and mobile applications are designed, i.e. which modeling languages, or more generically: which approaches, are used to support the design process. By design, we refer to activities conducted prior to technical implementation. The second objective of RQ1[2] is to find out how architectural knowledge about these applications is maintained. This includes decisions made during the architecting process, as well as information about the problem and solution space.

RQ2 originates from the conjecture that development teams partially reuse architectural design from previous projects. Architectural reuse improves development efficiency, which contributes to fast time-to-market of features. Furthermore, reuse is a means for risk mitigation [34]. Here, we want to find out which parts of an architecture are typically re-used.

The third question concerns the time span, developers expect their applications to reside on the market before they are discarded or subject to a major re-engineering. This is relevant because the cost-effectiveness of documentation effort is proportional to the expected lifetime of an application.

2.2.2 Methodology

A survey was conducted to collect data for our research questions. We chose to use a web-based questionnaire over individual interviews, because we wanted to reach a sufficiently large subject population. Questionnaire-based surveys additionally exhibit a higher degree of external validity than interviews [35]. When used with closed-ended questions and fixed response options, data gathered with questionnaires can easily be processed automatically. This is in contrast to interviews, which have high costs in terms of time per interview, traveling and processing the results. On the other hand, interviews provide greater flexibility and allow for more in-depth exploration of the respondents' answers. As described in Section 2, we plan to conduct interviews with a small

[2]Because of space limitations, in this paper, we bundled two of our original research questions into one.

number of the subjects at a later stage to get more in-depth insight in the phenomena we observe through the questionnaire.

We conducted a pilot study with three members from the target population to improve the wording, order and answering options of the questions.

2.2.3 Participants and Sampling

The target population of this study is professional software engineers who develop web and/or mobile applications. We used snowball sampling, i.e. the questionnaire was sent out to members from our professional network using e-mail. We asked the receivers to forward the questionnaire to colleagues and peers from their own network, which is a means to achieve a more randomized sample [36]. The questionnaire was distributed by an online form.

2.3 Data Analysis and Interpretation

This section presents the data analysis and interpretation. We primarily use descriptive statistics to analyze the collected data. The section is divided according to the three research questions. Table 2.1 maps the research questions to the questions from the questionnaire.

A total of 73 subjects responded to the questionnaire. From these respondents, 39.7% completed the questionnaire and answered all questions. For reasons of space limitations, we only report on the most interesting findings of our study.

The questionnaire has four sections. The first section (A) includes questions about the organization, role of the respondent and previous experience. The second section (B) concerns the applications developed. The third section (C) addresses the design, development and maintenance. The last section (D) concerns priorities regarding software development and software maintenance.

2.3.1 Participants and Sampling

The target population of this study is professional software engineers who develop web and/or mobile applications. We used snowball sampling, i.e. the questionnaire was sent out to members from our professional network using e-mail. We asked the receivers to forward the questionnaire to colleagues and peers from their own network, which is a means to achieve a more randomized sample [36]. The questionnaire was spread in form of an online form.

2.4 Data Analysis and Interpretation

This section presents the data analysis and interpretation. We primarily use descriptive statistics to analyze the collected data. The section is divided according to the three research questions. Table 2.1 maps the research questions to the questions from the questionnaire.

A total of 73 subjects responded to the questionnaire. From these respondents, 39.7% completed the questionnaire and answered all questions. For reasons of space limitations, we only report on the most interesting findings of our study. Our study database, which contains the questionnaire and all responses, can be found on http://2question.com/q1q3/.

The questionnaire has four sections. The first section (A) includes questions about the organization, role of the respondent and previous experience. The second section (B) concerns the applications developed. The third section (C) addresses the design, development and maintenance. The last section (D) concerns priorities regarding software development and software maintenance.

In the remainder of this section, we present the results and most relevant answers for every research question. Additionally we discuss results of supporting questions and control questions. This section ends with an interpretation of the results, discussion and expected and remarkable outcomes.

No	Question	RQ1	RQ2	RQ3
A1	What is the number of employees working in your organization?	✓		
A2	How many employees are working on Software Development in your organization?	✓		
A3	How many employees are working on your currently running project?	✓		
A4	Which of the following activities do you perform within your organization?	✓		
B1	What is the average number of users per day as anticipated at design time?	✓		
B2	What is the peak number of concurrent users during operations?	✓		
B3	Which of the following components are used in your web application?		✓	
B4	Which of these aforementioned components are obtained from a Cloud Service?	✓	✓	✓
B5	Suppose you start a new project or a major re-engineering of an existing application, which of these aforementioned components will you use again for Web applications that have been rebuild or evolved?		✓	
B6	What is the average lifetime of your application in number of years?			✓
B7	Within your organization how many applications share the same overall architectural design?		✓	
B8	How often do you release new features?			✓
C1	Which of the following activities are typical for software projects you have worked on?	✓	✓	
C2	Which of the following process methods are used in your projects?	✓		
C3	What types of tools do you use during your design process?	✓	✓	
C5	How do you ensure that knowledge about features, implementations, design decisions etc. is maintained?	✓		
D1	What are your three top priorities in software development?	✓	✓	✓
D2	What are your three top priorities in software development when you need to successfully maintain software in the long term?	✓	✓	✓

Table 2.1: Mapping of Questions and Research Questions

2.4.1 Analysis RQ1: How Are Web and Mobile Applications Designed and How is Design Knowledge About These Applications Preserved in the Industry?

The questions most relevant to RQ1 are "What types of tools do you use during your design process?" (C3) and "How do you ensure that knowledge about features, implementations, design decisions etc. is maintained?" (C5).

In question C3, we asked participants to specify the tools[3] used for design, the time they spend on each of these tools (as a percentage of the total time spent on design activities), and the quantity of results (number of occurrences or number of produced deliverables). The design approaches, where participants spend most time on are "Experimenting, building proofs of concept" (26%), "Documented concepts in written language like Word documents" (22%) and "Sketches like annotated block / line diagrams" (19%). With 11% of total design time, technical documentation (e.g. UML, SysML, ERD, Database models) received the lowest score. In terms of quantity, the top three answers were "Verbal communication" (14), "Sketches like annotated block / line diagrams" (6) and "Experimenting, building proofs of concept" (3).

For knowledge preservation (question C5), the top three methods used in terms of spent time (percentage of overall time spent on knowledge preservation) are "Documented concepts in written language like Word documents" (26%), "Documented code (with tools like JavaDoc, JSDoc or no tools)" (26%) and "Verbal communication" (17%).

[3]The term *tool* is used in a wide sense here, covering among others UML, free-text, but also conversations and informal whiteboard sketches

Additionally, we asked participants about their top three priorities during software development (question D1) and software maintenance (D2). During *development time*, the top three priorities are "Quality" (7,2%), "(Functional) requirements" and "time-to-market" (both 6,6%), and "Maintainability" (4,8%). During *maintenance*, the top three priorities are "Documentation" (18%), "Code quality" (17%) and both "Architecture" and "Maintainability" (6,8%).

2.4.2 Questions Related to RQ2: Which Parts of the Software Architecture Are Re-Used Across Modern Web Applications?

The most relevant question that relates to this research question is B5: "Suppose you start a new project or a major re-engineering of an existing application, which of these aforementioned components will you use again for Web applications that have been rebuilt or evolved?"

The top three results from C5 are "Webservice Application Programming Interface (API) (eg. RESTful, SOAP)" (86%), "SQL Database(s)" (83%) and "Server side web frameworks" (79%)

A supporting question is B7: "Within your organization how many applications share the same overall architectural design?" 76% of the applications share between 21% and 80% of the overall architectural design. This is equally distributed over the three categories. 21% are from applications that share almost all components. The rest (1%) does not share any component. Another supporting question is B3 where participants where asked about the types of software components they typically use in applications. The components mentioned most prominently (53%) are: *Build Tools*, *Test tools*, *Server Side Frameworks*, and *Web Services*.

2.4.3 Questions Related to RQ3: What is the Average Life Expectancy of Modern Web Applications?

The most relevant question related to this research question is B6: "What is the average lifetime of your application in number of years?".

62% of the applications have a lifetime between 1 and 5 years. 14% of the applications have a lifetime of more than 10 years. The lifetime of an application determines the selection of components. For start-up companies, the initial application architecture will be sufficient for the first period. When growing in number of customers, transactions, and processes, we expect that the initial application has to be replaced with a scaleable architecture and infrastructure.

2.4.4 Interpretation RQ1-RQ3

In this section, we discuss the results regarding all three research questions.

In many software engineering curricula, students are taught to use (semi-)formal modeling languages like UML for designing software before coding. In contrast to this, we found that technical documents are not intensively used for design purposes in the software engineering industry. Instead, at least for mobile and web applications, the design process is primarily driven by verbal communication and informal sketches. This is in line with Sonnenburg [37], who describes software design as a collaborative creative activity [37], which benefits from approaches that are not constrained by fixed notations and formalisms.

On the other hand, we found that projects create more output in the shape of technical documentation than in other forms. This may be surprising at first, as less time is spent on technical documentation. On the other hand, there may be a causal relationship between those two aspects, i.e. software engineers spend less time on technical documentation, because they are reluctant to spend time on non-engineering activities, i.e. activities that are no integral part of the built process.

In question C5, we assume a typical division between development and maintenance, in which developers in a project are not responsible for deployment and maintenance of applications. In this scenario, documentation is crucial for deployment and maintenance, as well as for managing responsibilities [38]. However, most participants chose "Verbal communication" as the primary method for handing over the code to other team members. In discussions with software engineers in the pilot group and remarks from participants, we found that engineers rather rely on proven practices in their teams, rather than formal methods, to design, develop and maintain applications. One of these proven practices is the use of verbal communication in weekly team meetings to discuss code and design. These discussions aim at improving the quality of the code by reviewing the contributions for that week and sharing the concepts and implementations.

In line with [39]–[41], we did not expect that webservice APIs (Simple Object Access Protocol (SOAP), RESTful) would be the most re-used architectural assets (question B5). We had rather expected that data

would have a higher value both for business and for software engineers and thus would be more often re-used that services.

With B6, we expected that the average life time of an application will be within 3 to 5 years (as in [42]). This is related to IT expenditures that are typically budgeted from capital expenditures. Capital expenditures have a typical amortization of 5 years. Nowadays, companies do not have to invest in costly server infrastructure anymore (capital expenditure). Instead, web and mobile applications are typically deployed in cloud environments, in which infrastructure is payed for as-a-service and is thus operational expenditure [43]. Furthermore, software engineers typically change their employer or job-role between 2 years [44] and 4.6 years [45]. Finally, software engineers typically favor building from scratch over brown-field applications that have been patched over the years. In the latter cases, the technical debt exceeds the cost of re-building from scratch.

2.5 Threats to Validity

In this section, we discuss possible threats to the internal and external validity of our findings. A common threat to internal validity in questionnaire-based surveys stems from poorly understood questions and a low coverage of constructs under study. The former threat was mitigated to a large extent by piloting the questionnaire with three participants form the target population. We asked these participants to fill in the questionnaire. Afterwards, they were asked to describe their interpretations of the questions and their answers. We used this input in multiple iterations to revise the questions and answering options. We addressed construct validity by explicitly mapping the questions of our questionnaire to the research questions (see Table 2.1) and by making sure that each research question is covered by multiple questions in the questionnaire.

External validity is concerned with the degree, to which the study results can be generalized for a larger subject population [46]. We used statistical methods to analyze whether our results are significant. Mason et al. postulate that, as a rule of thumb, questionnaires require between 30 and 150 responses in order to yield valid responses [47].

We had a total of 73 respondents; 39.7% of whom answered all questions. Thus, we suppose that the number of respondents is sufficient.

Two remarkable outcomes from the questionnaire (questions C3 and C5) are 1) that technical documentation is less popular than plain text documentation and 2) that continuity of knowledge is achieved primarily through verbal communication. We calculated the variance and standard deviation of our responses. For C3 the variance is 0.2 and thus very low; for C5 the calculated mean is 423, the standard deviation is 193 and the weighted value for verbal communication is 425. The actual weighted value deviates by 2 points only. Thus, the results with respect to our most surprising outcomes are statistically significant.

2.6 Conclusions and Future Work

In this paper, we investigated how web and mobile applications are designed and documented. We found that verbal communication and informal sketches are clearly preferred over modeling languages. To preserve and transfer application-specific knowledge, companies deem code documentation equally important as technical documentation. Furthermore, for many companies, verbal communication is the primary approach for transferring knowledge within teams. This may be surprising at first as it bears the risk that knowledge gets lost, because of key employees leaving the company or a lack of communication in teams. However, web and mobile applications are primarily developed in small teams using agile development processes. Such development approaches rely heavily on verbal communication, and practices like daily stand ups in Scrum achieve that knowledge is widely spread within the development team.

Another remarkable outcome is the very high degree of architectural re-use across projects. In particular, we found that web-service APIs, SQL databases and server-side frameworks are re-used across projects in more than 80% of the cases. This is certainly impacted by the focus of our research on web and mobile applications. Teams build up knowledge and expertise in certain technologies and exploit this knowledge to a large degree for reasons of efficiency.

Regarding the average expected life-time of web and mobile applications, we found that most applications (~60%) are built for being rather short-lived (1-5 years). Further investigation is required to understand the reasons for this phenomenon.

As explained in the introduction, we will use these results for creating an architecture framework streamlined for web and mobile applications. The framework will anticipate the reluctance to produce (semi-)formal documentation and the high degree of technological re-use. We will conduct further research to understand

the impact of the short life-times of such applications on the effort found reasonable for producing written documentation.

We started this research with a questionnaire to obtain quantitative data. The next phase in our research plan is to conduct interviews to collect qualitative and more in-depth data.

Chapter 3

Specification in Continuous Software Development[1]

Abstract The procession of Lean, Agile and DevOps development processes introduces new challenges and offers new chances regarding software specification and documentation. Challenges for instance because specifications, just like code and applications, are subject to continuous change; chances, because continuous software processes make use of a high degree of automation which also introduces efficient means for specification and documentation.
In this paper, we describe the continuous software design specification pattern, which contains guidelines and principles for specification in continuous development processes. In these processes, a software system is an evolution of life cycles where each iteration has a start, continuation and end of defining specifications. Therefore, the pattern explicitly distinguishes specifications to be created at the start of an iteration, specifications during an iteration, and a specification-refactoring at the end of each iteration. Apart from the pattern description, this paper describes the principles of continuous software development derived from lean software development, Agile, and DevOps.

Keywords Agile, Continuous Development, DevOps, Lean, Software engineering

3.1 Introduction

In our previous research, we have been investigating practices in design and documentation in web and mobile applications [30]. Our focus was on understanding the role of software architecture in these applications. Among other things, we tried to find out what is designed up-front (i.e. prior to implementation) and how. Among others, our results indicate that verbal communication plays a significant role in the preservation of knowledge. We also found that many companies struggle with the distinction between specification up-front and documentation afterwards. While specification and documentation are often seen as one, many approaches are either a good fit for specification or for documentation, but not for both.

The software projects, in which we observed these phenomena were predominantly governed using lean, agile or DevOps process models. The leading principle of lean software development [2] is to avoid efforts that do not increase value for the customer. Agile software development [1] tries to exploit the full potential of human collaboration in closely-interacting teams, thereby relying on short improvement cycles to achieve frequent delivery of working software. In DevOps [48], development teams are formed in way that members in each team cover the full set of competences, skills and responsibilities required to develop, operate, and maintain a software product. Because the development team is responsible for the entire product life cycle, it becomes more

[1] This work was originally published as:

T. Theunissen and U. Van Heesch, "Specification in Continuous Software Development," in *Proceedings of the 22ND European Conference on Pattern Languages of Programs*, New York, NY, USA, 2017, pp. 1–19. doi: 10.1145/3147704.3147709.

sensitive to operation concerns like security, scalability, performance, and portability. Additionally, DevOps aims at avoiding unnecessary transfers of artifacts between different teams, as such transfers usually require a significant communication and documentation overhead.

Lean, Agile, and DevOps all have certain principles in common that imply a paradigm shift regarding software specification and documentation: efficiency and effectiveness, learning, flexibility of the team, short iteration cycles, people skills, improvement and involvement of customer, and commitment of the organization. As opposed to many traditional software projects, which have a defined start and endpoint (which can be a point in time or a specific result), lean, agile and DevOps were designed to support continuous software development, in which continuity (i.e. the absence of a predefined end-point) is one of the major characteristics. This is primarily supported by rather short iterative development cycles.

In this paper, we elaborate specifically on the difference between specification and documentation in software projects that embrace the previously mentioned principles. The pattern CONTINUOUS SOFTWARE DESIGN SPECIFICATION differentiates specifications required at the beginning of an iteration, specifications required during an iteration and documentation of important results. Applying this distinction is a way of separating the concerns that developers have in specification. Not one size fits all, but instead, certain elements like information whiteboard sketches are only required temporarily, while other specifications need to last longer.

The rest of this paper is organized as follows: In Section 2, we describe the three process models lean, agile, and DevOps and identify principles they have in common. Section 3 describes the CONTINUOUS SOFTWARE DESIGN SPECIFICATION pattern. Finally, Section 4 identifies areas of future work.

3.2 Background

In this section, we present background work on agile software development, lean, and DevOps. These processes form the basis of what we later refer to as Continuous Software Development. Many of the described principles are enablers for a more lightweight way of software specification and documentation, which we describe in the pattern CONTINUOUS SOFTWARE DESIGN SPECIFICATION below.

3.2.1 Lean

Lean was originally developed as a manufacturing practice for cars. In the meantime, lean practices have been adopted by many other engineering discipline, among others by the software engineering discipline. Following Poppendieck and Poppendieck [2], Lean software development entails the following principles:

1. **Eliminating waste** Eliminating waste is the most fundamental lean principle. Waste refers to anything that does not produce value for a customer. Examples of waste in physical products are motion, transportation, and inventory. In software engineering, waste includes task switching of team members, defects (bugs), processes that do not have an immediate benefit, and paperwork.
2. **Amplify learning** This principle stems from the idea that development is rather a creative process than a systematic process. Developing software is a learning process with progressive insights, trial and error and reconsidering decisions based on tacit knowledge and implicit skills.
3. **Decide as late as possible** This principles, which primarily targets concurrent development of complex systems, advocates the exploration of decision options and delaying the final decision until it can be based on facts rather than speculation. In some situations, this may require building variation points into the system so that development can continue while decisions are postponed.
4. **Deliver as fast as possible** Deliver as fast as possible is required for fast time to market. Customers like fast delivery. For software development, this often translates to more flexibility. In the first place, this may seem contradictory to Decide as late as possible. In the reality, it rather complements this principle. While the former principle causes decisions to be delayed, the latter principle makes sure that decisions are nevertheless made frequently. In combination, this means that decisions are delayed to the last possible moment (with a release being the last possible moment).
5. **Empower the team** Empower the team by trusting the capabilities of an experienced team. Decisions should be made inside the team and not be imposed on the team. As a consequence, teams need a certain level of maturity. It is not easy to assemble a team that is both experienced and has junior developers, has a lot of knowledge and is also willing to and capable of learning.
6. **Build integrity in** Users, customers, and developers all just see one aspect of a software product. The aspect of integrity aims at one integral system where perceived and conceptual integrity is built-in. The perceived integrity is about user experience and tries to anticipate future use cases. A software system has integrity, if it has a coherent architecture, high usability, is maintainable, adaptable, and extensible.

7. **See the whole** People who are experts in a specific area of software engineering tend to maximize the performance of the part of the software they are most knowledgeable about, while loosing sight of the system as a whole.

Many of the above principles can be related to specification and documentation efforts. The most prominent one being Eliminating waste, as specification and documentation that does not provide an immediate benefit and that on contrary even causes rework effort for keeping it in-sync with the system is considered waste.

3.2.2 Agile

In 2001, the agile manifesto [1] gave rise to a new way of thinking and collaborating in software projects. Since then, several process models (the most prominent being Scrum [49] and XP [50]) evolved that embrace the principles postulated in this manifesto:

1. Customer Satisfaction Satisfy the customer by delivering software early and continuously.
2. Welcome changes Anticipate frequently changing requirements.
3. Frequent releases Deliver working software in short iteration cycles.
4. Collaborate with business people Collaborate with business people daily.
5. Trusted Individuals Form teams of motivated people and trust them to get the job done.
6. Face-to-face conversation Face-to-face communication is most efficient and effective.
7. Working software is progress Measure progress by the amount of working software developed.
8. Sustainable pace Processes should be sustainable in a way that the team can keep up pace.
9. Technical excellence Good design and technical excellence enable agility.
10. Simplify Favor simplicity over complexity. Avoid unnecessary work.
11. Self-organizing teams Empower teams to manage themselves.
12. Regular adjustments The team reflects on process regularly to become more effective.

Likewise lean software development, many agile principles relate to specification and documentation. The principles Face-to-face conversation and Trusted Individuals for instance express that verbal communication between skilled individuals is better than communication via specifications and documentation. Additionally, the agile manifesto even explicitly promotes ”Working software over comprehensive documentation”, which is a direct hint towards the amount of documentation required in agile projects.

3.2.3 DevOps

The term DevOps, coined in 2009, is a concatenation of Development and Operations. The following principles were derived from a literature study on DevOps, conducted in 2014 [48]. Culture The primary characteristic of a DevOps culture is increased collaboration between the roles of development and operations [51]. Another important element is shared responsibility. Likewise agile and lean, the DevOps culture advocates an organizational shift to autonomous teams, who strive for a continuous improvement of their process. Additionally, Walls [52] emphasizes open communication, alignment of incentives and responsibilities, respect, and finally, trust. Automation A cornerstone of DevOps is a high degree of automation. Automation facilitates the other characteristics of DevOps. Typical automated steps in a CI/CD pipeline are agile development, integration, delivery, deployment and operations. Measurement DevOps promotes the introduction of reliable measures to get hold on the development process. [HP 2016] mentions four dimensions of metrics that should be covered in any DevOps process: velocity, quality, productivity, and security. This principle is covered in the solution where at the finish of an iteration, the evaluation is described. Evaluation implies a set of measurements. Sharing In DevOps, sharing refers to knowledge, tools and successes [53]. Sharing knowledge in a DevOps team is the basis for efficient collaboration. Sharing coding styles, development tools and implementation techniques to develop features and maintain environments and infrastructures are key to be and stay successful. Teams should also share success, e.g. by celebrating important releases together. Services The principle of services represents the trend that software companies are moving from a product model to a services model. Key characteristics for services are intangibility, inventory, inseparability, inconsistency and involvement [54]. Quality assurance Team needs to build quality into the development process. Because iterations are short, the new code is brought easier and faster into production. This includes cross-functional concerns such as scalability, performance and security. To increase quality, it is required that both developers, operations and customers have a close relation to have a better understanding of issues, enablers or risks. Furthermore, monitoring processes, including development metrics and end-user actions, enables early detection of problems [55]. Structures and standards DevOps is not just a team issue, but requires standards that the whole organization embraces. Shifting to DevOps is a major organizational effort that requires commitment from all participating parties. Culture is the DevOps principle that has the greatest impact on specification. This principle embraces the standards, values,

and ways of working that are manifested in collaboration, shared responsibility, and autonomous teams. These DevOps values lead to less or at least loose specifications.

3.2.4 Principles of Continuous Software Development

In this section, we revisit the principles of lean, agile and DevOps to extract a common set of principles that among others have an impact on the way teams deal with specification and documentation practices. In the remainder of this paper, we will refer to development processes that exhibit these shared principles as CSD. The principles are:

1. **Efficiency, effectiveness** In continuous software development, one strives for an optimal balance between efficiency and effectiveness. Effectiveness refers to the desired outcome, i.e. the percentage (quality) that the result matches the objectives. Efficiency means the amount of resources (money, time, people) used to realize the results. Furthermore, there are two limitations related to effectiveness and efficiency. First, resources like time, money and people, are limited. Second, even adding virtually unlimited resources to a project could not force desired results. Brooks states that adding people to a project takes time to become productive, adding people increases communication overhead, and there is a limited divisibility of tasks [56][2]. Because of these limitations and dependencies, there is a trade-off between maximizing effectiveness and maximizing efficiency. The ambition is to achieve as much as possible for both efficiency and effectiveness without losing the balance. Regarding efficiency, the primary means in lean is eliminating waste. Effectiveness refers to delivering working software, achieving customer satisfaction, and simplicity. Additionally, measurements are required for checking if development and operations are on the right track. Both, efficiency and effectiveness should strive for a sustainable pace.
2. **Learning, improvement** Learning and improvement are about progressive insights; and planned and unplanned improvements. The objective is to continuously improve the development process as well as the learning outcome. Therefore, perform regular feedback sessions like e.g. a sprint retrospective in scrum, encourage learning by doing and learn from mistakes. The process models achieve this by promoting short feedback loops, sharing ideas, uncertainties and mistakes, and a culture of trust.
3. **Flexibility** Flexibility is about possibility and willingness to adapt to new situations. The objective is to benefit from actual insights and agreements. Therefore, teams need to welcome changes and establish a culture that embraces uncertainties and last minute changes and is able to think out of the box. There is no necessary requirement for learning with flexibility. It might well be possible that one trivial situation must be changed for another trivial situation. The core message of flexibility is the ability to adapt to new (unforeseen) situations.
4. **Time-to-market** Time-to-market refers to short delivery cycles or frequent releases. The objective is to deliver features as fast as possible. Improvements will start earlier and there is a better fit between end-user, customer, organization, and development team. To accomplish this fast TTM, a high degree of automation from development to deployment is required.
5. **Trust, attitude** Trust and attitude refer to: 1) the trust given to the team and 2) the team's attitude to show that they are worth the given trust. This trust is reciprocal; all parties should trust and live up to the given trust. The objective is to let everyone excel in their competences, but also to think outside the box by involvement from other parties. This requires a specific organizational structure and standards. Typically, these types of organizations have little management with a high degree of autonomy for the teams.
6. **Competences** Competences refer to highly skilled people who are experienced in a wide range of technology. The objective is to build teams capable of bringing together the concerns from the team, the customer, and the organization. Within the team, there should be a shared and coherent view on the software product. Additionally, quality management processes are required to make sure that competences and capabilities match or exceed the requirements.
7. **Competitive advantage** This refers to the risk that people tend to excel in a specific skill while at the same time losing sight of the big picture. Face-to-face conversation between parties and team members reduces the risk of losing sight. As part of this view, teams should focus on delivering added value, while leave commodity solutions to service providers. The objective is achieve a competitive advantage by focusing on core competences and outsource commodity services.
8. **Involvement** This includes involvement from end-user, customer, developers and operations. The objective is to share common goals. This requires shared principles and priorities, and understanding of

[2]"If one woman delivers a baby in nine months, then nine women can't deliver a baby in one month"

one's concerns and standards.

In appendix A we show a table that maps the principles of continuous software development to Lean, Agile and DevOps.

3.3 Pattern: Continuous Software Design Specification

This pattern describes a lightweight manner to deal with specifications in a continuous software development process.

3.3.1 Context

You have deliberately chosen to apply the principles of continuous software development. Your team has worked together on multiple software products. The people in your team know each other well and have an established communication culture. The team members are also knowledgeable about the technological domain, in which the software product to be developed resides.

You have already settled a proven build pipeline, which includes for instance a set of build tools (e.g. Maven[3] or Gradle[4]), a distributed version control system (like Git[5]), a document management system and wiki (for instance Confluence[6]), and a task and project tracking tool (see for instance Jira[7]).

3.3.2 Problem

The developers in your team strive for omitting the creation and maintenance of artifacts that are not immediately required for building a high-quality software product. You consider maintaining a specification document beyond the realization that provides just another view on a software product that is already specified by the source-code itself as wasted effort. Totally omitting specification, however, comes with certain downsides:

- Specifications are needed as a basis to reason and communicate about architectural challenges.
- Specifications are needed as input for task-planning activities, e.g. for setting up a work breakdown structure.
- Specifications are required to settle agreements regarding interfaces between modules developed by different team members, or other teams working on other parts of a larger software system.

Thus, the problem is: *How to provide just-enough adequate specifications for reasoning about architectural problems, supporting planning activities, and defining interfaces between team members?*

3.3.3 Forces

The following forces need to be considered:

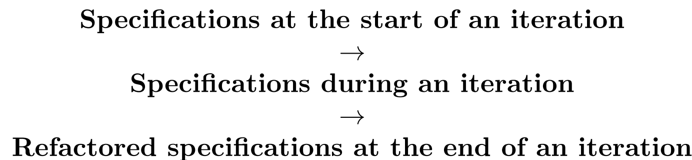
1. **Shaping thoughts** The process of specifying contributes to a better understanding of the problem and envisioned solution of an application.
2. **Progressive insight** During a software development process, developers continuously gain new insights that they want to or need to consider in the implementation. As a specification is a kind of implementation plan, new insights either need to be woven into the specification before they are implemented (which can be seen as wasted effort in lean terminology), or the implementation derives from the specification.
3. **Specification gaps require assumptions to be made** Things that are not explicitly specified (i.e. written down) require assumptions to be made by developers during the implementation. Silent assumptions bare the risk that individual team members make decisions that interfere with or even contradict each other.
4. **Hidden disagreement** Related to the previous force, relying primarily on oral communication bears the risk of hidden disagreement. That is, developers discuss a problem or an envisioned solution and actually talk across purposes without realizing.

[3] https://maven.apache.org
[4] https://gradle.org
[5] https://git-scm.com
[6] https://www.atlassian.com/software/confluence
[7] https://www.atlassian.com/software/jira

5. **Overestimated competences and underestimated complexity** People tend to overestimate their own competences and skills [57]. This leads to an underestimation of the problem complexity. When extensive specification is omitted prior to implementation, the real complexity of the software problem may be uncovered only piecemeal during the implementation, which may lead to significant rework.
6. **Time to market** Ever increasing demands for faster time-to-market require faster deployment and therefore shorter development cycles. Often, there is no time for extended technical specification upfront and complete documentation afterwards.
7. **On-boarding of new team members** When new team members enter the development team, or the entire software product is transferred to or picked up by a new team, the software design needs to be transferred to the new people in charge.
8. **Explored design space** The required coverage degree and formalism of specifications relates to the degree to which the design space of the application is already explored by patterns, frameworks, libraries and other assets. Applications which can be built upon existing frameworks or high-level programming languages, for instance, require less specifications than applications in domains that are not (yet) covered by such assets. In those cases, the abstractions introduced by the frameworks, templates and libraries, form a vocabulary for developers that allows them to communicate more efficiently and they induce structure to software systems that can be understood by studying the framework rather than having to study the application built upon the framework. Other examples are (parts of) applications that need to interact closely with custom hardware and or custom communication protocols. These applications may also require a higher degree of specification.

3.3.4 Solution

Instead of aiming for providing a single self-contained and comprehensive specification document, align your specification process with the continuous development process. Therefore, *split the specifications created into three different types of specifications that serve different purposes and address different concerns:*



In the evolution of a system, each iteration has a life cycle where specifications are created at the beginning of an iteration, altered during development, and some specifications become obsolete at the end of an iteration. Specifications are not deleted, but are kept in a repository without further intervention, unless deletion is part of a specification refactoring. Only those specifications that are relevant to the next iteration or maintenance survive an iteration.

We refer to specification as an artifact created prior to or during the realization of a piece of software. Specifications do not necessarily cover a whole system, but they can also concern a small part of the system (e.g. a part required for the implementation of a user story). Here, specification is a prescription meant to support and constrain the implementation. Documentation, on the other hand, is a description of the actual implementation for preserving and sharing rationale and knowledge about the implementation. Documentation is just another deliverable, if it is valuable, doing it is not free and probably displaces something else, e.g. development time. In the following sub-sections, we describe each of those types in detail.

Specifications at the start of an iteration.

Specifications required to effectively start an iteration (e.g. a sprint in a scrum-project) should include the following items:

1. A list of requirements to be addressed in the iteration.
2. An architectural vision.
3. A description of the technological ecosystem in which the software will be developed.
4. Information about the most important architectural concerns (i.e. quality attribute requirements and business drivers), which determine the priorities of decisions to be made.

It is not advisable to aim for providing a comprehensive specification covering the aforementioned aspects in all detail; instead the specification should be deliberately limited to information required to start a development iteration thereby taking into account the skills, knowledge and experience of the development team. Some of

the mentioned artifacts may already exist when they were created in a prior iteration. In such cases, the artifacts are only revisited and adapted if required.

Requirements to be addressed in an iteration (1) are typically captured in a task planning tool. It is not advisable to duplicate requirements, i.e. to also specify them elsewhere.

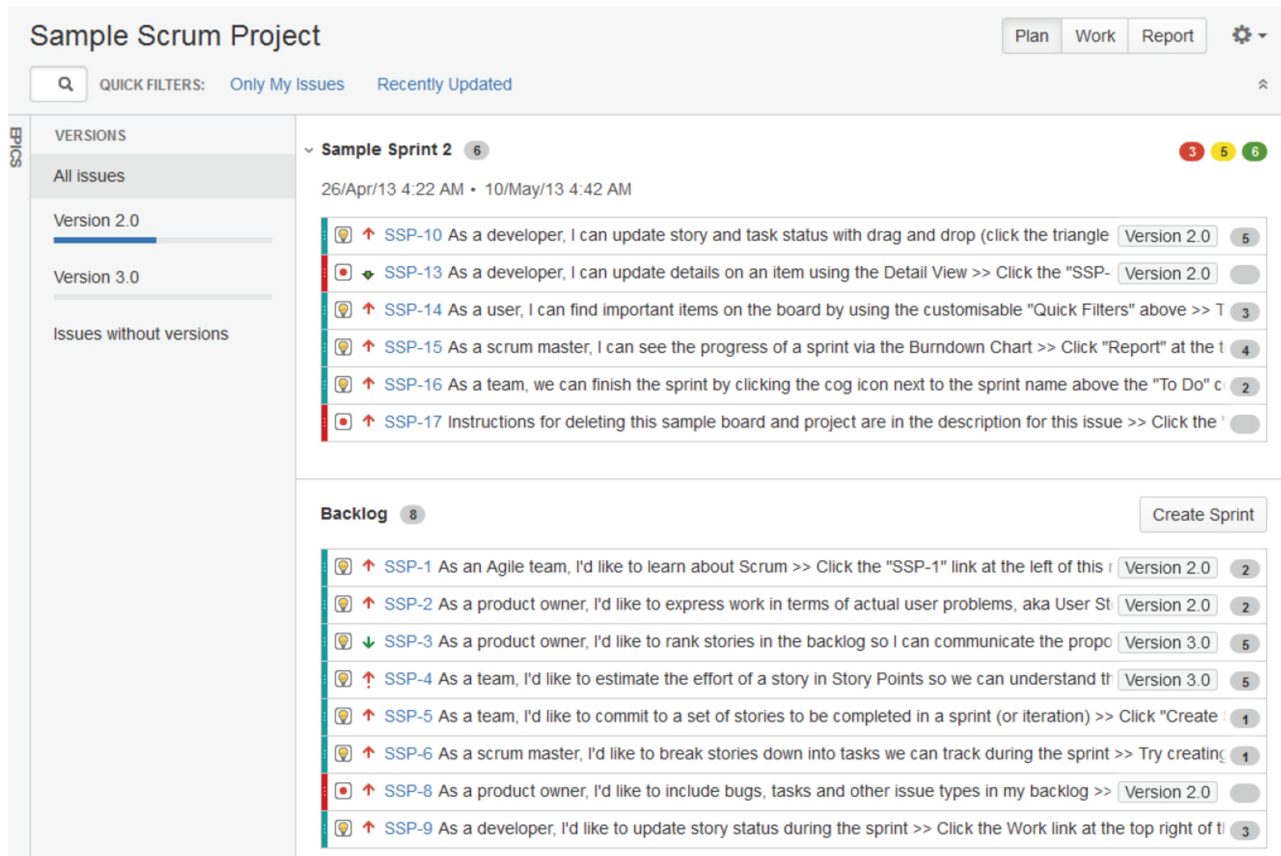


Figure 3.1: A product backlog in Jira [58].

Figure 3.1 shows a product backlog created on a scrum-board in Jira. Requirements to be addressed in the next iteration are shown underneath the heading *Sample Sprint 2*. Underneath this so called *Sprint Backlog*, the *Product Backlog* contains a list of all other requirements to be addressed in the future. The product backlog is continuously maintained and must be seen as a living artifact that always shows the current state of potential requirements. Note that the requirements captured here are usually primarily functional requirements, or even concrete development-related tasks derived from functional requirements. We will get to non-functional requirements below.

The second and third items required to start an iteration are an architectural vision and a description of the technological ecosystem. Often, the space available on a regular whiteboard is sufficient for this kind of specification. Figure 3.2 shows an example.

The whiteboard shows a specification created by a scrum-team at the beginning of Sprint-0. The team created this specification together to settle agreements required to start with the first development iteration. The example contains both of the aforementioned items:

High-level architectural vision The whiteboard shows an informal sketch of a three-tier layered architecture with a mobile application client and a web-application, a restful service facade, and a relational database. Sub-systems are not specified formally, but in a way that is sufficient for the team members to understand each other.

Technological Ecosystem The whiteboard sketch also gives hints regarding the technological ecosystem, in which the system will be developed. In this case, the back-end uses Java technology combined with open-source database management systems, a cross-platform framework for the mobile application, and a client-side JavaScript framework for the development of the web application. Note that some technologies are marked with a question mark, which indicates that a final decision has not yet been made. However, the diagram provides enough information to get an idea of the target ecosystem. Subsequent decisions can be made during the development iteration itself.

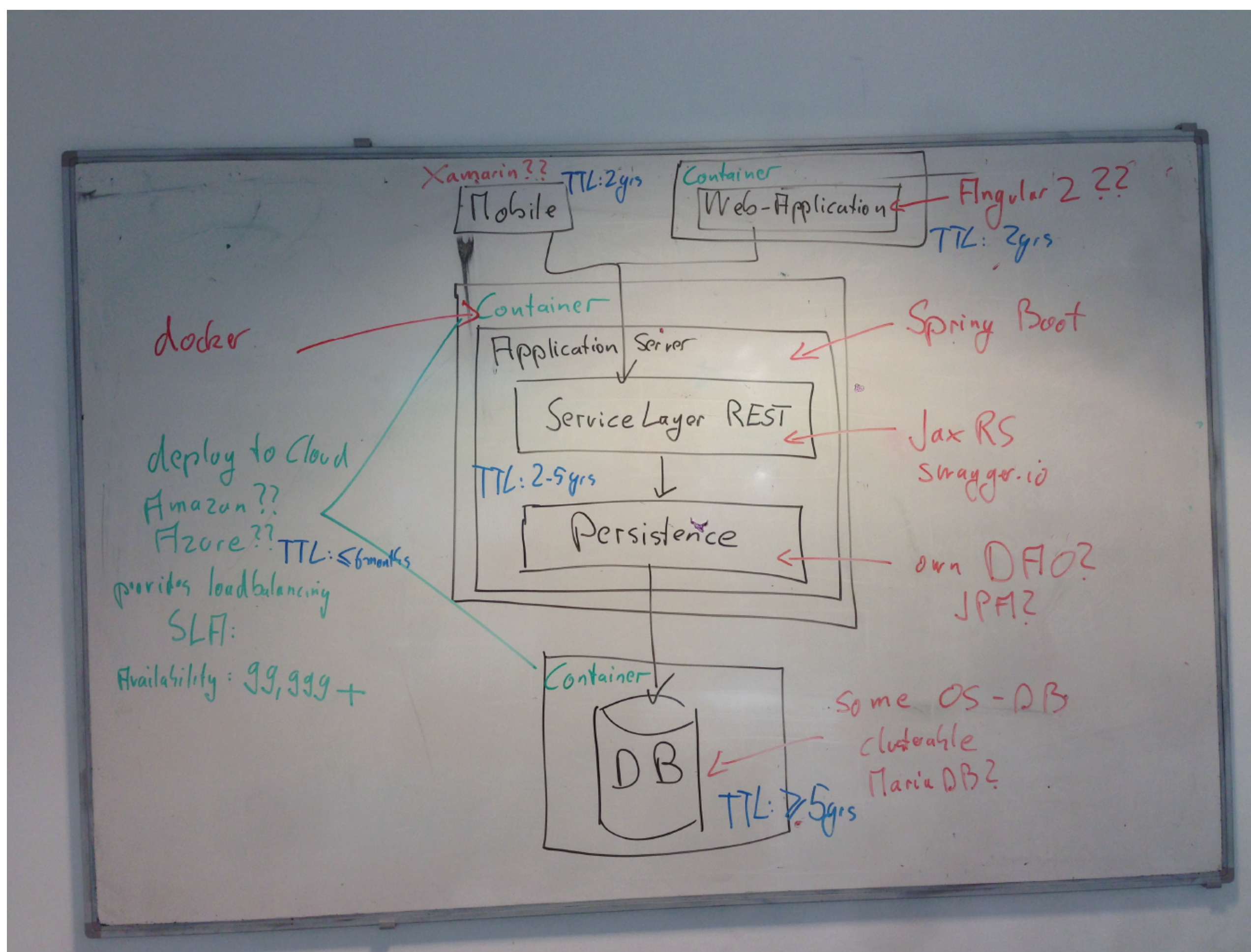


Figure 3.2: Whiteboard drawing of architecture vision.

The fourth and final item required to start an iteration is information about the most important architectural concerns to be considered during the iteration. Usually, agile or lean teams focus on functionality while assuming typical requirements regarding quality attributes. Only in situations, in which special requirements exist that derive from the typical needs of the type of application developed (in which the team is experienced), the quality attributes and other architectural concerns are captured. In this case, Figure 2 only mentions a few important concerns in a very informal way. The different envisioned sub-systems will be provided as containers, which need to be deployed to a cloud service. The whiteboard mentions a target availability of 99,999%, which needs to be guaranteed by the cloud provider. Furthermore, the diagrams shows the envisioned life-time (TTL in the diagram) for the different parts of the system. As no more concerns are mentioned, the team assumes typical concerns for mobile and web application regarding performance, scalability, security and the like.

Specifications during an iteration.

Specifications to **continue** development during an iteration cycle should codify agreements made between individual team members or between multiple teams working on the same larger application.

Examples of such specifications include, but are not limited to (RESTful) API specifications, data models, user interface specifications, and (architectural) design decisions which need to be considered by stakeholders other than the decision maker her or himself. Examples of such decisions are the technological choices mentioned on the whiteboard shown in Figure 2. Of course, as this pattern concerns continuous development, these specifications are continuously adjusted if required to serve the aforementioned purpose.

As mentioned above, the specifications are not part of a single self-contained document. You should rather create each specification artifact where it is either naturally used as part of the software development process, or it is automatically generated and updated from something that is part of the development process. We show two examples that demonstrate this principle of a single source of truth. The first example is an API specification in Yet Another Markup Language (YAML). The second example concerns tests written with

Cucumber[8]. Both specifications are stored in a git repository.

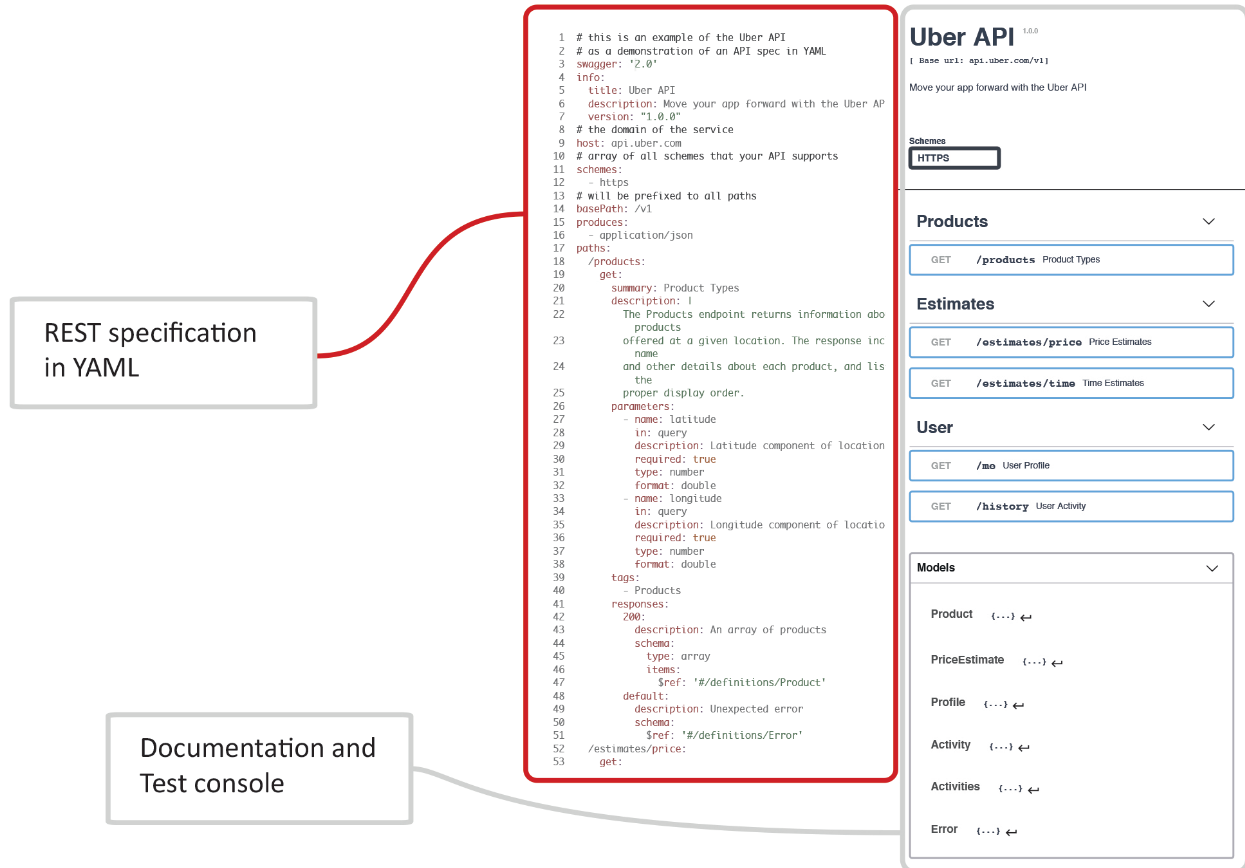


Figure 3.3: REST API definition with swagger.io.

Swagger.io[9] is a tool for defining a REST API, including (required) input parameters, output parameters, types of parameters and descriptions. The description language of the API is provided in YAML directly within the source code of the application. From this API specification in YAML, the tool generates code for testing or further development. Developers can always rely on this API specification as it is the single source of truth. Figure 3.3 shows an example of swagger in use. So the code itself serves as specification.

Another example for a tool using the same principle is Cucumber. Cucumber is a BDD [59] test tool. BDD was preceded by TDD [S60]. In these types of development, tests are the specifications for the development team.[10] Cucumber is used for behavior-driven development in automated acceptance tests. Typically, a team with customers, developers and testers explore the area, each from it's own perspective and competences. After clearing up misunderstandings and explicating assumptions, this results in a set of specific examples that are typical for the problem domain. An example of a feature description in Cucumber is shown in Figure 3.3.

The principles also apply for many other types of artifacts generated using specific tools (e.g. UMLtools, database management systems, or UI-frameworks). Always strive for creating the specs in the tool, which is also used for the development and do not duplicate artifacts. If however, no tool is available that automatically generates more readable versions of such specifications, then the source code itself should be used as a specification.

A special role is taken by decisions made by developers during the iteration. Many of these decisions are primarily relevant for the developer himself and do not necessarily need to be shared with other team members. Examples are decisions about design patterns (assuming they do not have architectural implications) used, libraries that do not induce implications for modules developed by other developers, or design principles applied by the developer to structure the code. Certain decisions, especially those having architectural impact,

---

[8]https://cucumber.io/
[9]http://swagger.io/
[10]We focus on the test as specification and do not present merits and (dis)advantages of test-first development methods

travels.feature × Setup.java × NS.java ×

```
Feature: travel from to a station

  Scenario: travel with two children
    Given I am at www.ns.nl
    And I clicked accept in cookie popup
    And I choose to travel today
    And I fill "Utrecht" as the station Vanaf
    And I fill "Amsterdam" as the station Naar
    And I select "1e" as the Klasse
    And I select "1" as the number of Reizigers
    And I fill initials "J.J.G.W.M." for traveller "1"
    And I fill name "Bakker" for traveller "1"
    And I fill birth date "22-11-1988" for traveller "1"
    And I have clicked on InWinkelwagen
    And I have clicked on Railrunner
    And I have clicked on DirectBestellen
    And I fill to travel today
    And I select "2" as the number of Children
    And I fill "J." as the first child initials
    And I fill "Bakker" as the first child name
    And I fill "22-11-2010" as the first child Birth date
    And I fill "R." as the second child initials
    And I fill "Bakker" as the second child name
    And I fill "02-03-2009" as the second child Birth date
    When I have clicked on KidsInWinkelwagen
    Then the total price is "17,80"

```

Figure 3.4: Feature description in Cucumber for a Dutch train travel website.

should be specified and shared with relevant stakeholders. Candidates for such decisions are architectural styles, patterns and tactics and other decisions that have a significant impact on quality attributes and externally visible interfaces of the system. Not all of these decisions must be documented though. Only document decisions that are non-obvious to the team and that cannot be recovered from artifacts already created as part of the development process. Examples of decisions that do **not** have to be documented per se are used frameworks and third part libraries, if such information can be easily retrieved from a build file (e.g. a pom.xml file from Maven), for instance. As a rule of thumb you should spend documentation effort primarily on decisions that were hard to make, caused a lot of discussion, seem counter-intuitive, were significantly impacted by people external to the team, and also on decisions which turned out to be not good after the implementation. One way of documenting decisions in a lightweight manner is using specific architecture decisions views [S61], [62].

Figure 3.5 shows an example of a decision-relationship-view [S61], which shows relationships between decisions made and their status. A relationship view in combination with a decision-forces view [S61]. Figure 3.6 captures sufficient information of decisions to support the team during the iteration, in which the decisions are made.

Finally, as specification items are spread over multiple different locations, it is advisable to provide an overview page which contains links to the diverse locations of the specification items. A natural place for such an overview page is the team's wiki. Note again that information should not be duplicated on the wiki. Instead the wiki should contain links to locations that do not change frequently. Otherwise the risk remains that the information on the overview page gets outdated quickly. Figure 3.7 shows an example of such an overview page created in Confluence.

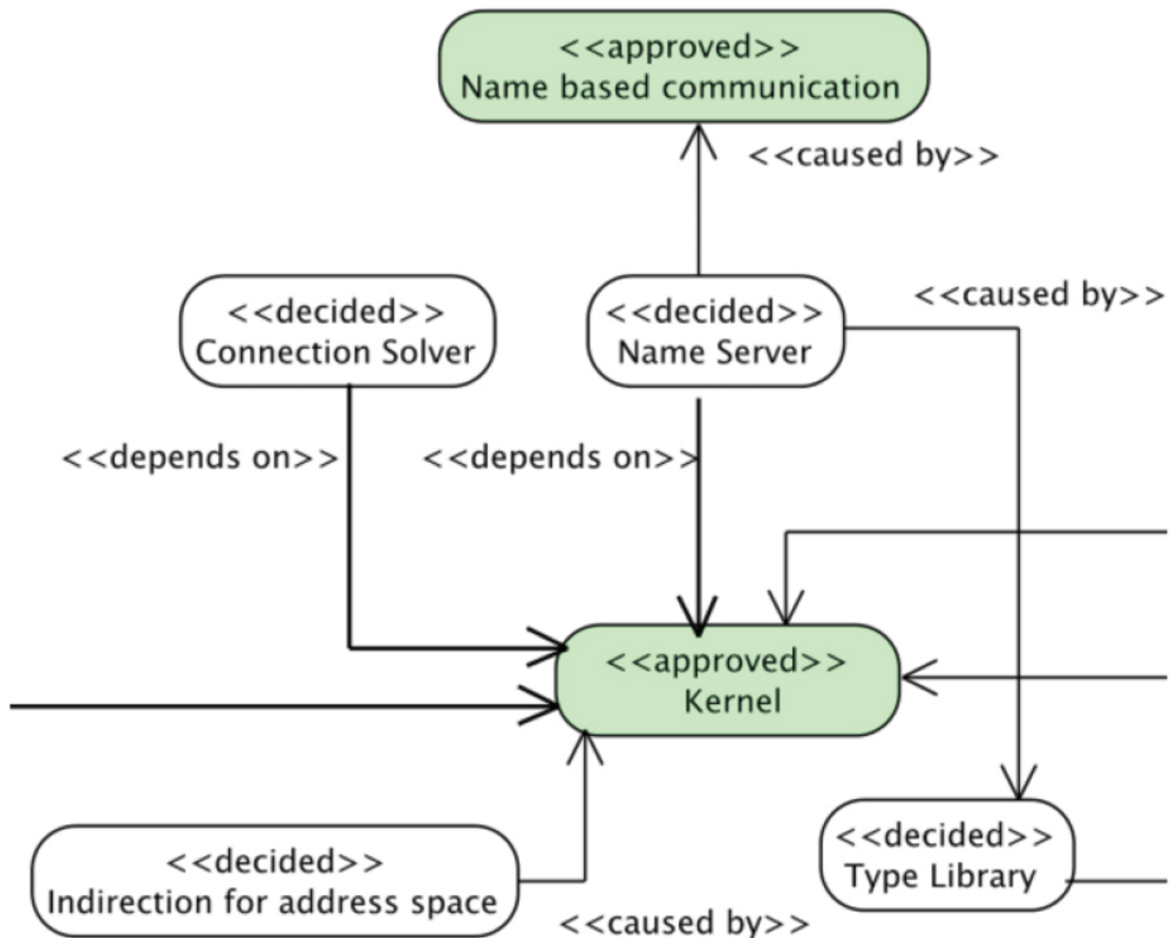


Figure 3.5: Example of a decision relationship view.

Refactored Specifications at the End of an Iteration.

At the end of an iteration, you should revisit specifications created and updated during the iteration and decide explicitly on items relevant

Fig. 5: Example of a decision relationship view

Fig. 6: Example of a decision forces view

for the next iterations. The process we propose here is roughly comparable to a refactoring process for source code. It can be seen as a kind of specification refactoring. During this process, all artifacts that exclusively serve documentation purposes are revisited and either kept unchanged, simplified, or thrown away. Specification items that could be kept unchanged are documented decisions that are still valid or an architectural vision that is still valid. Some specification artifacts can be simplified or made more concise with the knowledge a developer gained during the iteration. Examples of artifacts that can be thrown away are UML-or box-and-line diagrams of software parts that were fully implemented in the meantime. In such cases, the code itself is often a better and more accurate specification of the system than the diagrams could be. This is the same as the architecturally-evident coding style, used by [63].

Fig. 7: Overview page with links to specifications relevant during a specific iteration

Especially the documented decisions should be revisited as the status of decisions frequently changes throughout an iteration. We suggest to cleanup the decision views and only keep those decisions that are still in a decided state. Furthermore, important decisions or decisions that required long discussions should be described in more detail, for instance using a decision detail view from van Heesch, Avgeriou, and Hilliard [S61]. This view is independent of a specific iteration. It should be up-to-date at the end of each iteration. Additionally, as iterations in agile or lean teams are often accompanied by one or more releases, specifications

Decision Forces			View technology			Data storage	Middleware	DBMS	
			<decided>	<discarded>	<discarded>	<decided>	<discarded>	<discarded>	<decided>
			Java Swing	PHP	JSF	Central DS	EJB	MySQL	PostgreSQL
Architecture significant requirements									
Code	Description	Concern(s)							
R1	Avg. response time <= 0.1s	Time behavior	++	+	-	-	-	+	+
R5	Integrate mult. payment providers	Extendability	+		+		+		
R6	Reliability of data storage	Reliability				++	+	+	++
R8	Availability of full service (99.9%)	Reliability	++	+	+	+	-	+	+
R9	Support growing no of users	Scalability	++	+	-	-	+	-	?
R13	Security (personal data protection)	Security	+		?	+		?	?
R16	Client platform independence	Portability	+	++	++				
R23	Operability of user interface	Usability	++	+	+				
R24	Communication via Internet	Network comm.		++	++	+	+	+	+
R26	HBCI support	Banking protocols	+	?	+		+		
R27	No paid 3rd party licences	Development costs	+	+	+			++	++
Other forces									
F1	Inhouse experience	Development time							
F1.1	Swing (very good)	Development time	++						
F1.2	PHP (decent)	Development time		+					
F1.3	JPA (good)	Development time							
F1.4	MySQL (very good)	Development time				+		++	
F1.5	JSF (very good)	Development time			+				
F2	Strategic knowledge development	Competitiveness							++
F2.1	Learn Postgres	Competitiveness				+			++
F2.2	Improve Javascript skills	Competitiveness	--	+	+				
F2.3	Learn Jquery	Competitiveness	--	+	+				
F4	Linux server available	Development costs		+	+	+	+		+
F5	Non business criticality	Business criticality							+
F7	Resource usage on server	Resource utilization	++	-	--	--	--	+	?

Figure 3.6: Example of a decision forces view.

should cover a description of (automated) steps to test, deployment and operations. Again, make use of the principles mentioned above. It is better to point to a provisioning script on a wiki, rather than describing a deployment process verbally. Along with every release, the current state of all documentation artifacts should be kept, e.g. to cope with situations, in which multiple versions of a software are used by customers.

3.3.5 Consequences

In the following, we will discuss the consequences of applying the CONTINUOUS SOFTWARE DESIGN SPECIFICATION pattern.

1. **Shaping thoughts** The process of specifying contributes to a better understanding of the problem and envisioned solution of an application. When applying the pattern, developers discuss the envisioned architecture of the system and the technological ecosystem typically using a whiteboard. The whiteboard sketch only serves as a means to support the discussion. It is not meant as a documentation. As a consequence, the whiteboard sketch becomes less and less useful the more time passes. This effect is deliberately accepted here. The iteration start specifications are meant only to enable a quick-start to the iteration and to support planning activities.
2. **Progressive insight** At any time during the iteration, specifications are only created or updated as part of the development process. New insights can always be considered. The pattern embraces changes over following a plan.
3. **Specification gaps require assumptions to be made** The solution described by this pattern advocates a radical reduction of specifications to a minimum. Naturally, this causes specification gaps which force the team members to either silently assume problem or solution-related aspects, or to explicitly discuss them with their team members. As mentioned in the context section, this can only work well if the team is experienced and has an established communication culture. There is thus a correlation between the amount and detail of specifications needed and the experience and skills of the development team. Likewise, this applies for hidden disagreement. Typically, agile and lean process models address these problems by weaving regular retrospective sessions into development iterations, in which the team -among other issues also discusses their communication and conflict-management strategies.

Continuous Specs Sprint 3

Created by

This page contains links to specification items relevant to sprint 3. Add or updated links to specification items that are relevant to multiple team members.

Development Pipeline

- Jenkins: http://ci. .nl/jenkins/job/ASD%20A%20Passenger%20Service%20-%20Develop
- Stash develop branch: http://git. l/stash/projects/ASD1617S1A/repos/passenger_service/browse
- Sonar: http://ci. l/overview?id=13561

Technical Specs

- Restful API Specs: http://ci. .nl/swagger/project/ASD%20A%20Passenger%20Service%20-%20Develop
- Relational Schema Customer_DB: http://mysq nl/schemeGenerator?id=13561
- Virtual Machines and Container configuration: http://proxmo .nl

Task and Issue Tracking

- Scrumboard: http://jira. l/secure/RapidBoard.jspa?rapidView=394&view=planning.nodetail

Decision Views

- Decision Relatinship View: http://decision .nl/relView?id=1761
- Decision Forces View: http://decisions nl/forcesView?id=1761

Like Be the first to like this

Figure 3.7: Overview page with links to specifications relevant during a specific iteration.

4. **Overestimated competences and underestimated complexity** As a consequence of no big upfront specification, the complexity of software problems and solutions is regularly underestimated. The same holds true for the developers' competences. Therefore, this pattern should only be applied in teams using agile or lean principles, which rely on short iterations, review and retrospective sessions. Hidden complexity is therefore typically discovered and discussed regularly and the team can learn from previous mistakes and new insights.
5. **Time to market** Using the pattern, the team does not spend effort on documentation that does not provide an immediate benefit for the current iteration. It is thus beneficial for achieving shorter release cycles and within this quicker time to market.
6. **On boarding of new team members** When applying this pattern, an offline preparation of new team members is drastically hampered. This is not so problematic for new team members entering an established team, as new team members can be brought slowly up to speed by taking part in the regular team ceremonies and picking up simpler tasks in the beginning. For transferring an application to an entirely new team, the specifications advocated by this pattern are not sufficient. However, the information documented can serve as a basis for providing a more comprehensive team transformation document which provides more detail on the architecture, important decisions made and the overall design of major components.

3.4 Outlook

This paper presents a first effort to describing specification processes for continuous software development projects. The CONTINUOUS SOFTWARE DESIGN SPECIFICATION pattern can be applied in a context where a team already built up specific knowledge and skills, e.g. about the development process or the domain of an application. In the future, we plan to document another pattern that describes how these necessary preconditions can be achieved by a development team. This pattern will cover knowledge about technology, knowledge about processes and agreements that need to be made by a team. This includes a way of dealing with tacit knowledge [64] the developers have. Among others, the pattern to be documented will make use of

templates, frameworks and libraries to enable continuous development. The processes, context and environment for this second pattern are described in continuous development principles.

3.5 Acknowledgments

We would like to thank our shepherd Uwe Zdun for his critical feedback and useful hints during the shepherding process of EuroPLoP 2017. We would also like to thank Allan Kelly for doing an extensive review of this paper after the conference.

3.A. Appendices

Principle of ConSD	Lean	Agile	DevOps
1. Efficiency, effectiveness One strives for an optimal balance between efficiency and effectiveness	**1. Eliminating waste** Waste is anything that does not produce value for a customer. The primary focus for eliminating waste is minimizing resources while achieving the same results.	**1. Customer satisfaction** **7. Working software is progress** Both customer satisfaction and working software relate to results for the customer. This includes short iterations for faster TTM and reducing risks. **8. Sustainable pace** Strive for maximum effectiveness and efficiency in short iterations. This avoids the common phenomenon that goes along with long iterations where teams start slowly (thus inefficient) and get hasty when deadlines are approaching. The latter comes with the downside that quality requirements are neglected and technical debt is accepted to deliver within the deadline. **10. Simplify** Simplification relates to efficiency by focusing on a) a minimum viable product; b) limiting work in progress, strive for minimizing backlog items; c) eliminate waste by avoiding extra features, partially done work; d) minimize organizational structure by reducing the number of roles, ceremonies, etc. "Maximizing the amount work-not-done" can be achieved by coding (standards, templates, libraries), architecture (be specific in architecture, do not strive for generic and 'beautiful' architecture), testing (small unit-tests), automation (CI/CD pipeline), and standards (common, shared, proved, easy to understand)	**3. Measurement** In DevOps, the development process is monitored using process- and team-performance-related measurements. On the one hand, the measurements provide a profound basis for direct process improvements. On the other hand, these improvements can be evaluated by analyzing their impact on the respective measurements in retrospect.
3. Flexibility Ability to adapt to new, unforeseen, and possible trivial situations.	**3. Decide as Late as Possible** The exploration of decision options and delaying the final decision until it can be based on facts rather than speculation. This implies uncertainties for the team as long as a final decision is not taken. The team needs to be flexible to handle these uncertainties	**2. Welcome Changes** The team welcomes changes to give the customer a competitive advantage.	**Structures and Standards** These are the *defacto* and *de jure* values, standards and behaviors that contribute to last minute changes. *De jure*: what is agreed upon, either by law, code or conduct or agreements. *Defacto*: what actually happens. These two concepts do not necessarily exclude or include each other. In some companies, the policy is to deploy whenever a change is committed (Amazon deploys 50 times a day) where other companies have a policy for regular releases (Microsoft's "patch Tuesday").

Continued on next page

Table 3.1 – continued from previous page

Principle of ConSD	Lean	Agile	DevOps
4. TTM The lead time it takes from concept to minimal marketable product.	**4. Deliver as Fast as Possible** Customers like fast delivery. Therefore, Lean strives for frequent and fast delivery.	**3. Frequent Releases** This includes minimum viable product (MVP) as well as minimum marketable product (MMP). Agile processes strive for delivering software increments after each iteration.	**2. Automation** Automation in the CI/CD pipeline includes testing, integration, delivery, deployment and operations. The automated steps from development to deployment make delivery fast, repeatable, and predictable. Furthermore, automation leads to managing and finally reducing risk by optimization of critical steps.
5. Trust, Attitude All parties trust each other and live up to the given trust.	**5. Empower the Team** Decisions are made inside the team and not imposed on the team.	**5. Trusted Individuals** Support the team with trust.	**1. Culture** The culture of trust can be found in the while of *defacto* and *de jure* values, standards and behaviors within your organization.
6. Competences Highly skilled people who are experiences in a wide range of technologies.	This principle is compatible with the Lean mindset, but not explicit in the Lean philosophy.	**9. Technical Excellence** The development team employs skills, as well as process-related skills. **11. Self-organizing Teams** The competent team has clear objectives of what to achieve but limited rules on how to achieve these objectives.	**6. Quality Assurance** Defining characteristics that refer to the desired outcomes, i.e. the percentage(quality) that the result matches the objectives. The team is well aware of the quality of the result and process that is required and act according the standards.
7. Big Picture The risk that people tend to excel in a specific skill while at the same time losing sight of the big picture.	**7. See the Whole** The risk of not having an overview, is that it may create an environment where suboptimal behavior occurs with suboptimal results. Seeing the whole, or big picture, reduces suboptimal solutions as the big picture embraces the individual suboptimal, or or internal competitions.	**6. Face to face Conversation** Face-to-face conversation between stakeholders and development team reduces the risk of losing sight. The risk of a best solution for a single developer that is not supported by other developers and stakeholders is mitigated by a continuous dialogue between developers and stakeholders.	**5. Services** Teams focus on delivering added value, while leaving commodity solutions and non-core competences to others, e.g. service providers.
8. Involvement Shared principles and priorities, understanding of one's concerns and standards.	This principle is compatible with the Lean mindset, but not explicit in the Lean philosophy.	**4. Collaborate with Business People** Understand each other's needs, possibilities and weaknesses.	**7. Structures and Standards** There are the *defacto* and *de jure* values, standards and behaviors within an organization, like e.g. a code of conduct. The structures and standards within an organization encourage involvement, e.g. by training knowledge-meetings, or tine to experiment (like Google's[11] and Atlassian's 20% rule[12]).

Table 3.1: Principles of CSD mapped to Lean, Agile and DevOps.

[11] https://abc.xyz/investor/founders-1etters/2004/ipo-letter.htmI
[12] https://www.atlassian.com/blog/archives/20_time_experiment

Chapter 4

Software Specification and Documentation in Continuous Software Development – A Focus Group Report[1]

Abstract We have been observing an ongoing trend in the software engineering domain towards development practices that rely heavily on verbal communication and small, closely-interacting teams. Among others, approaches like Scrum, Lean Software Development, and DevOps fall under this category. We refer to such development practices as Continuous Software Development (ConSD). Some core principles of ConSD are working in short iterations with frequent delivery, striving for an optimal balance between effectiveness and efficiency, and amplify learning in the development team. In such a context, many traditional patterns of software specification, documentation and knowledge preservation are not applicable anymore.
To explore relevant topics, opinions, challenges and chances around specification, documentation and knowledge preservation in ConSD, we conducted a workshop at the 22nd European Conference on Pattern Languages of Programs (EuroPLoP), held in Germany in July 2017. The workshop participants came from the industry and academia.
In this report, we present the results of the workshop. Among others, we elaborate on the difference between specification and documentation, the special role of architecture in ConSD in general, and architecture decision documentation in particular, and the importance of tooling that combines aspects of development, project management, and quality assurance. Furthermore, we describe typical issues with documentation and identify means to efficiently and effectively organize specification and documentation tasks in ConSD.

Keywords Agile, Continuous Development, DevOps, Lean, Software engineering

4.1 Introduction

In the last decade, we have been observing a shift in the software industry towards software development practices that rely on verbal communication, closely interacting small teams, shorter planning and controlling horizons (often called sprints or iterations), and frequent delivery. Popular approaches include Scrum [49],

[1]This work was originally published as:

U. Van Heesch, T. Theunissen, O. Zimmermann, and U. Zdun, "Software Specification and Documentation in Continuous Software Development: A Focus Group Report," in *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, New York, NY, USA, 2017, p. 35:1—-35:13. doi: 10.1145/3147704.3147742.

Lean Software Development [2], and more recently DevOps [51]. We refer to collections of these practices as CONTINUOUS SOFTWARE DESIGN SPECIFICATION [S31] to take into account their predominant characteristics of continuous, short and time-boxed iterations and incremental refinement. In continuous software development, some traditional patterns of software specification, documentation and knowledge preservation are not applicable anymore. One of the main challenges is dealing with software specifications and documentation in a continuous development flow. Specifications take many forms, e.g. requirements specification, architecture specification, design specification, test specification, and deployment descriptors. Team members primarily value specifications that have an immediate benefit for their own tasks during an iteration (e.g. specifications of interfaces between sub- systems). However, depending on the type of software and the type of specification, these specifications may need to serve additional needs. For instance, they may need to support reasoning about technical risks, help stakeholders understand complex systems, support (offline) communication between stakeholders, or capture design decisions with long-term impact. In our ongoing work on continuous software development (see for instance [S31]), we conjecture that the diverse types of specifications and documentation with different lifespans, different levels of formalism, different levels of detail, and different forms of codification

handled individually and differently so that they can satisfy the diverse needs of stake- holders in CONTINUOUS SOFTWARE DESIGN SPECIFICATION. To further explore relevant topics, opinions, challenges and chances around specifications and documentation in CONTINUOUS SOFTWARE DESIGN SPECIFICATION (with a focus on architectural specifications), we organized a workshop at the 22nd European Conference on Pattern Languages of Programs (EuroPLoP), held in Irsee, Germany in July 2017. In this paper, we report on the results of the focus group and describe directions for future work on this topic. The rest of this paper is organized as follows: Section 2 explains the setup of the focus group and the characteristics of the participants. In Section 3, we present the results of the discussion. Finally, Section 4 presents directions for future work on this topic.

4.2 Workshop Setup and Participants

The workshop took place in shape of a so called focus group, for which the EuroPLoP conference has reserved time slots of 90 minutes. We announced the focus group prior to and during the conference to attract participants interested in specification in CONTINUOUS SOFTWARE DESIGN SPECIFICATION. Participation was voluntary and advance registration not required. To attract peoples' attention and to scope our own areas of interest, we posed the following set of initial questions as part of the focus group announcement:

1. What is the difference between specification and documentation? What purposes do they serve and for whom?
2. What is the role of architecture specification in continuous software development?
3. Should software specifications be organized around self-contained documents?
4. What is the role of models in this context?
5. What alternate forms of organizing specification items could be a better fit for continuous software development?
6. What is the life cycle and scope of the different types of specifications and how does this life cycle relate to the agile life cycle, for instance?
7. Refactoring has become a common technique for improving the structure and quality of source- code. How can similar techniques be used for specifications?
8. What is the role of (architectural) design decisions in this context? How to share, document, or update them?
9. How to efficiently preserve (architectural) knowledge and skills a development team gained through- out multiple iterations/projects? What difference exists between generic and application-specific knowledge and skills?
10. How can agile practices (e.g. sprint retrospectives) be leveraged in this context?
11. How can information needs by external reviewers and auditors be satisfied?
12. What is the impact of business and technical domain specifics in this context?

Additionally, we presented a poster summarizing our previous work on specification in CONTINUOUS SOFTWARE DESIGN SPECIFICATION (see Appendix A).

4.2.1 Participants

In total, we had 16 participants partly from industry and partly from academia. To find out more about the background of our participants, we asked them to fill in a questionnaire in the beginning of the focus group. The

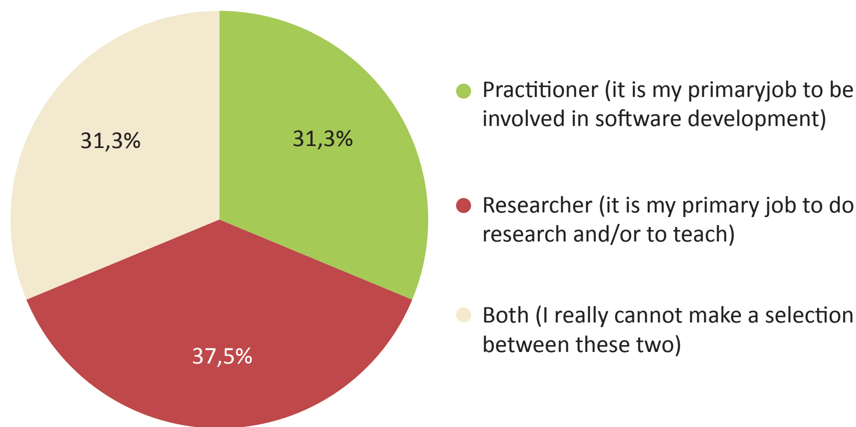


Figure 4.1: Affiliation of participants

questionnaire and the results are available online[2]. As Figure 4.1 shows, roughly one third of the participants consider themselves as practitioners, the other participants are either academics or related to both industry and academia. We also asked the participants about the number of years in software engineering experience (see Figure 4.2). With an average of 13 years of experience, we mainly had senior software engineers in the workshop. Figure 4.3 shows the tasks, our participants are usually involved in when doing software projects. Most participants cover a wide range of typical software engineering tasks.

4.2.2 Setup

After a short introduction on continuous software development, we split the participants into four break-out groups. Each break-out group was moderated by one of the authors. The initial set of questions shown above served as a rough question guide. However, the discussion was not constrained by these topics. Each break-out group discussed for one hour and noted insights on a flip-chart. Afterwards, the entire group met again to discuss the findings. In the following, we summarize the findings of the groups.

4.3 Results

We discussed the difference between specification and documentation; terms which are often used interchangeably in the software engineering domain. Documentation, as understood by our participants, is a piece of writing conveying information about a software artifact, to be consumed after the software artifact was built. The primary purpose of documentation is to help stakeholders better understand certain aspects of the system, be it because they need to support or further develop the system, approve certain aspects, review or audit it – or because they pay for it. Specifications are seen as presentations of information meant to support the development process of a software artifact. Specifications are created and consumed *before or during* the development of a software artifact. In many cases, specification artifacts are also used as documentation. These cases can be particularly problematic in CONTINUOUS SOFTWARE DESIGN SPECIFICATION, because specification artifacts are only maintained while they are necessary (or at least immediately beneficial) for the realization process of a software artifact. As a consequence, the actual realization regularly derives from the last maintained state of the specification. When being used as-is as documentation, these artifacts contain inconsistencies and specification gaps and are thus less usable for the purpose of documentation. In the following sections, we present additional insights about documentation and specification gained during the focus group.

[2] https://goo.gl/47NtNf

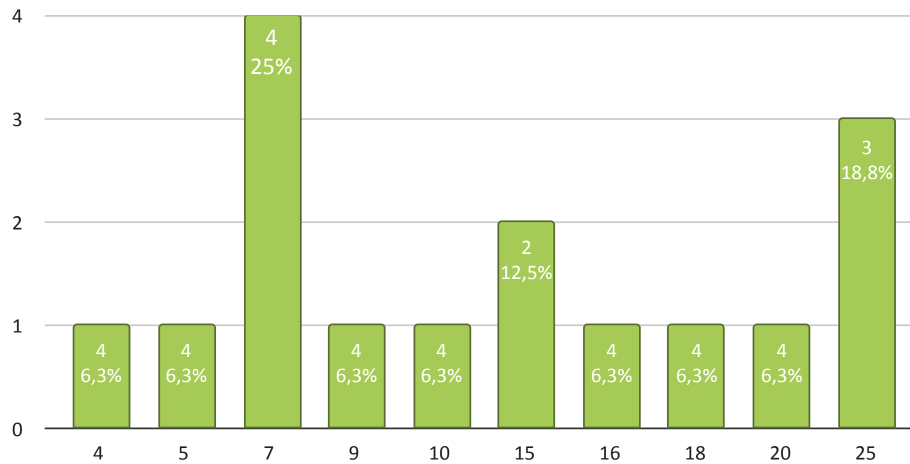


Figure 4.2: Years of Software Engineering (SWE) experience

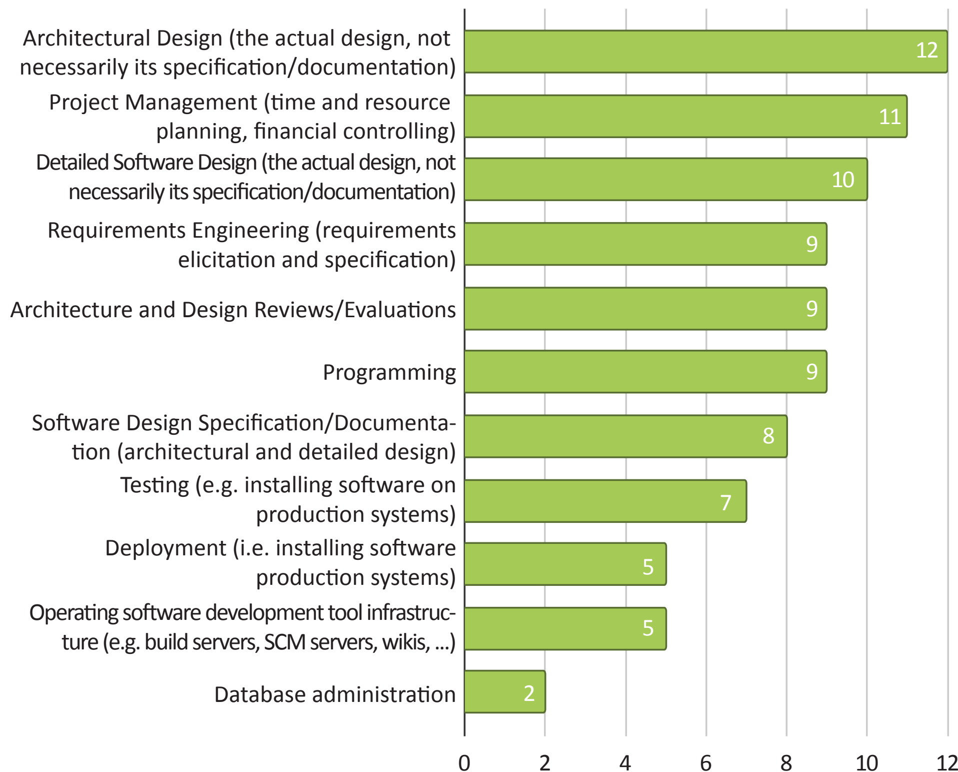


Figure 4.3: Involvement in SWE tasks

4.3.1 Specification

Specifications are created upfront and give instructions, rules and guidelines on how software artifacts should be built or what properties they should have. They are usually prescriptive. Specifications are described as "living artifacts", as they are continuously adjusted with progressive insights, as long as they have a benefit for

the development process. Different types of specifications exist, for instance requirements specifications, test cases, user interface specifications, architectural specifications, or design specifications (e.g. UML diagrams).

4.3.1.1 Levels of formalism and detail

We found that the degrees of formalism and completeness vary greatly between projects and artifacts. In CONTINUOUS SOFTWARE DESIGN SPECIFICATION, team members work together intensively, know each other well, and rely on oral communications. Therefore, informal rich pictures (e.g., white board sketches) are preferred over formal specifications. Formalisms are only used when being required by tools used (e.g. test specifications in Cucumber2), or because very detailed interface specifications are required (e.g., between a hardware team and a software team or between the provider and the consumers of a public API) [65]). Related to this, teams develop a common body of knowledge, which consists of knowledge about specific technologies, patterns, and solutions applied in the past. Teams explicitly or tacitly refer to this common body of knowledge when creating specifications by leaving out details that external consumers of the specifications would require to form a complete picture, or by using a vocabulary that can be understood by the team members only. An example of the latter is the use of the term "request controller" in a specification, which if seen out- of-context, is so generic that it is meaningless for external people, while members of the team know exactly which component in an existing system is meant by the term. Thus, teams who do not know each other well and do not share a common body of application-specific and application-generic knowledge, require more formal and more detailed specifications than teams who share such a body of knowledge. Experienced teams can apply a kind of specification-by-exception approach, in which only derivations from their standard way are specified. One participant stated it like this: "We know what we are doing, so no need to write everything down.". This phenomenon, in our perception, is independent of the type of specification.

4.3.1.2 Architecture specification

One break-out group explicitly discussed the role of architecture specification in CONTINUOUS SOFTWARE DESIGN SPECIFICATION. At first, the idea of creating comprehensive up-front specifications may seem to conflict with the principles of CONTINUOUS SOFTWARE DESIGN SPECIFICATION, such as avoiding waste and being agile. Nevertheless, the members of the focus group emphasized the important role of architecture reasoning and architecture specification at the beginning of larger projects or re-engineering efforts. Apart from the often-cited advantages of architecture like identifying risks and reasoning about quality attributes, architecture (primarily smart system partitioning) is seen as an enabler for agile working. This is predominantly the case for larger systems which are too complex to be handled in the working memory of a human being at the same time. Smaller systems, and/or systems built the same way as other systems in the past, do not require much architecture specification. One participant said: "Let's not create large systems!" to express that systems that do not require architecture specifications to be manageable, are a much better fit for CONTINUOUS SOFTWARE DESIGN SPECIFICATION. Architecture issues arise mainly in large systems. If large systems are required by the nature of the problem, then architectural styles that split the system into small parts that can be developed and comprehended in isolation can be a solution. Microservices are one example of such an approach [65]. In CONTINUOUS SOFTWARE DESIGN SPECIFICATION, architecture specification is often done using a whiteboard and primarily describes the system partitioning into sub-systems and major components. For each component, the responsibilities and interfaces are described. Architectural design is discussed during iteration planning meetings (e.g. a sprint planning meeting in Scrum) and if required in a special "architectural stand-up" meeting during iterations.

4.3.1.3 Tooling

Especially in CONTINUOUS SOFTWARE DESIGN SPECIFICATION, teams rely heavily on integrated tool suites that combine aspects of project management with features of development tools. An example of such a stack is the combination of an IDE (e.g. IntelliJ[3] for Java), Git[4], JIRA[5] , Confluence[6], Jenkins[7], and SonarQube[8]. With such a tool stack, specification activities, coding, testing, quality management, and project controlling

[3]https://www.jetbrains.com/idea/
[4]https://git-scm.com/
[5]https://www.atlassian.com/software/jira
[6]https://www.atlassian.com/software/confluence
[7]https://jenkins.io
[8]https://www.sonarqube.org/

are closely intertwined and enable lightweight traceability between the different activities. To give an example, imagine the following typical flow (using tools mentioned by the participants):

1. A functional requirement is specified in form of a user story with acceptance criteria and an effort estimate in a backlog in JIRA.
2. When a team member picks up the requirement, (s)he discusses design implications with other team members using a whiteboard.
3. The team member takes a picture of the whiteboard and shows it on a page in confluence, which furthermore contains agreements (e.g. on interfaces) made with the team members.
4. Directly in JIRA, (s)he then creates a feature branch in the git repository and starts coding in the IDE. The IDE has integrated git support. (S)he uses the analysis features of the IDE to achieve high test coverage and to make sure the code conforms to the previously agreed on coding standards, which are also checked by the Continuous Integration (CI) server, Jenkins in this case.
5. The time spent on the issues is (semi-)automatically logged by JIRA, so that (s)he only needs to approve the efforts when logging work on an issue.
6. When the feature is readily implemented and tested, (s)he pushes the code to the git repository. The push triggers the CI-server, which runs tests, checks test coverage and coding standards; furthermore, the CI server triggers the code quality service (here SonarQube) and reports the results to the developers.
7. Afterwards, the developer creates a pull-request in git, which triggers her team mates to do a code review, supported by the analytics provided by the SonarQube dashboard.
8. After the pull request was merged with the master branch, the developer marks the JIRA task as done. A burn-down chart is automatically updated to reduce the remaining required efforts in the current iteration.

The process sketched above shows how development activities are closely accompanied and sup- ported by the tool chain. As a side effect, the tool chain provides traceability between requirements, tasks, design artifacts, code, required time, and code quality. Apart from supporting the realization of software artifacts (here a functional requirement), the information in the tools also serve documentation needs, as one can easily click through the history of executed tasks. The process above entails the following forms of specification:

- A requirements specification in form of a user story.
- Acceptance criteria, which likewise serve as acceptance tests.
- An effort estimation for the user story.
- A design specification in form of a white board sketch and additional agreements between developers specified in Confluence.
- Test specifications in the source code.
- Coding standards to be used in the IDE and in the CI-server.
- Further code quality standards to be used by SonarQube; some of these standards are architectural (e.g. regarding reliability, security, maintainability, and complexity).

Apart from the design specification on the white board and the Confluence page, all specifications are required and used by the tools, which apply the specifications automatically to support the developer.

While they are key artifacts for systems built in a model-driven way [66], architecture specifications still seem to play a tangential role at least in some CONTINUOUS SOFTWARE DESIGN SPECIFICATION communities. That said, code quality tools support some forms of architectural analysis, as mentioned above. However, some popular examples exist of tools that use architectural specifications in the same way as the specifications described above. Ansible[9], for instance, is a tool for automating deployment tasks using so-called play- books: text-based specifications of how production, staging, test, and development machines are set-up. Apart from automating deployment tasks, Ansible can be used for checking standard compliance, e.g., regarding security measures.

4.3.2 Documentation

As described above, we define documentation as artifacts meant to be consumed after a software artifact was realized. Especially in CONTINUOUS SOFTWARE DESIGN SPECIFICATION, the need and the effort spent on documentation is a sensitive topic, because developers strive for avoiding efforts that do not provide immediate value. Additionally, it can be stated that the vast majority of software developers does not like creating documentation [67]. The focus group discussions related to documentation dealt with the following questions:

[9]https://www.ansible.com

1. For which purpose do we create documentation and what types of documentation do we observe in CONTINUOUS SOFTWARE DESIGN SPECIFICATION?
2. When should we create documentation and how much is enough?
3. What is the role of architecture decision documentation in CONTINUOUS SOFTWARE DESIGN SPECIFICATION?
4. What problems can be observed regarding documentation?
5. How to effectively and efficiently provide documentation in CONTINUOUS SOFTWARE DESIGN SPECIFICATION?

In the following sections, we summarize the results for each of those questions.

4.3.2.1 Purpose and types of documentation in continuous software design specification

Generally, documentation is meant to explain aspects of software artifacts to stakeholders. Consequently, different types of documentation exist to address the diverging information needs of the stakeholders. Basically every type of specification mentioned above has a corresponding type of documentation. As one example, design specification is used to support the initial creation of a software artifact, while design documentation is used to explain the design of a realized artifact to developers who need to maintain or further develop the software artifact. One participant stated that “places with rapid staff turnover more urgently feel the need for elaborate documentation”. Additionally, for some types of applications, documentation is required for getting market admission (e.g. for medical or safety-critical systems), or documentation is a contractual deliverable.

As in the context of specification, the participants said that missing documentation is most problematic in large systems (without further discussing what large means exactly). Smaller systems could usually be comprehended by analyzing configuration files and source code. Regarding architecture documentation, a participant stated that “architectural design documentation is not needed very often during the development. When it is required, you pull the architecture.”. Pulling here refers to generating architectural overviews from other existing artifacts, e.g. UML component diagrams from source code.

To support the automatic generation of design documentation, the participants in different break- out groups mentioned the tool Doxygen[10]. Doxygen is an example for a tool that creates documentation from source code and special annotations used in source code. Apart from providing textual explanations on modules, methods, parameters and the like, Doxygen can generate several graphs and diagrams to explain the structure of the software in terms of modules and packages. This approach is compatible with the mindset of the software craftsmanship school of thought, who proclaim that the truth is only in the code [68].

4.3.2.2 When is documentation created and how much is enough

As described above, documentation is meant to be consumed after the realization of a software artifact. After, in this context, can mean during a subsequent task, in the next iteration, as part of an approval process by stakeholders, or during maintenance, for instance. Most participants explained that they use items created as specification also as documentation. In such cases, they plan for extra time close to releases for updating specifications so they are self-contained and consistent with the current state of the software. Documentation is treated as a piece of technical writing that needs to use the language of the prospective consumers. Especially in cases where documentation is a contractual deliverable, documentation tasks are taken over in the task planning and controlling system, efforts are estimated, and the results are reviewed to increase the quality of the writing. However, the participants agreed that the time spent on documentation must be limited to the minimum responsible amount. For reasons of efficiency, one participant advised: “Do slightly less than you think is required”, so that if stakeholders complain that information is missing, this can easily be added, but no time is unnecessarily spent on documentation.

4.3.2.3 Architecture decision documentation

Architecture decision documentation is a special type of documentation. The literature advocates thorough documentation of architecture decisions, e.g. using decision templates (e.g. [69], [70]) or dedicated decision views (e.g. [S61], [71]). During the focus group, the participants agreed that some form of architecture decision documentation is needed. However, the detail level and required comprehensiveness of architecture decision documentation was discussed controversially. Some participants stated that only the outcome of decisions is documented, but not the rationale behind the decisions and not the considered alternatives. One participant

[10] https://www.doxygen.nl

explained that he would "not document rationale because it is transient.". What he meant was that decisions are made in a specific context that can quickly change over time. Part of the context is the knowledge and experience of the people who made the decisions, the available technologies at that time, current software hypes, and approaching dead- lines, to name a few. He concluded that the rationale would most likely not be (fully) valid anymore and that therefore the effort you would need to spent on thoroughly documenting rationale is not reason- able. Another participant added that "having rationale too explicit can also cause people not to think carefully themselves". These statements of course could be seen as self-serving assumptions, because the participants might be developers themselves who do not enjoy creating documentation.

Interestingly, the participants mentioned that most decisions are documented in the beginning of iterations, while other types of documentation are delayed to the last responsible moment where the documentation needs to be delivered. Especially for architecture decisions, it would be more beneficial to document them after the software artifacts were realized and evaluated, because this gives new insights regarding the fitness of the decisions, which could be processed in the documentation. This is particularly the case for decisions that would not have been made with the progressive insight available.

4.3.2.4 Recurring problems of documentation

We also discussed typical problems with documentation and identified the following recurring issues (or "documentation smells" to pick up a term from refactoring). Participants reported incidents such as:

- **Version maze.** Often, documentation is not explicit about the version of the software it describes. It is then unclear to the reader whether the information provided is still up-to-date. Even worse, documentation is sometimes compiled of items that describe different versions. In such cases, the documentation drastically looses merit.
- **Cyclic references.** Documentation items often consist of multiple items or documents referencing each other intensively. Sometimes, important aspects are not thoroughly described, because documentation items reference each other in a cyclic way without actually providing the information at some place.
- **Incompleteness.** The participants experienced that documentation often contains "TODOs" and gaps, i.e. parts of the system, or aspects, that are not described. This could be related to the phenomenon that many people write documentation incrementally rather than iteratively. The writer starts with a high level of rigor and comprehensiveness, but eventually misses time or motivation to continue in the same fashion. As a result, the documentation becomes more and more inaccurate, or contains large gaps. When wikis are used, empty pages or page stubs with "under construction" as only content are a symptom of this issue.
- **Copy/paste of code or raw specification.** Sometimes, documentation writers take over large large snippets of code or design artifacts that are too detailed to effectively serve as documentation. One participant gave an example "A large class diagram with hundreds of classes, methods and parameters thrown at the reader has no value". Sometimes, production data ends up in external documentation by accident; usage of text that violates Netiquette or is not politically correct has also been reported and qualifies as a specification/documentation smell as well.
- **Presentation planet.** Very often, documentation is prepared in form of a slide-based presentation. This can easily be explained by the fact that managers and other non-technical stakeholders appreciate short summaries with a language they understand. Using slide-based presentations as the only means to document comes with several downsides. Presentations are often way too abstract for many purposes (especially for developers and operators), the slides alone are ambiguous and require the "voice-over" to be understood, and they typically do not contain references to the realized items they document. Info decks, as described by M. Fowler[11]can be a reasonable compromise.
- **Zombie specifications.** Also known as dead documents. Specifications and documentation items might not have any readership and might not have been updated in a long time. You can tell from missing references to them in meetings and other specification/documentation items, as well as from access logs. Such items qualify as waste from a lean management point of view; they should either be updated and improved to meet an information need in a particular target audience, marked as "stalled" and archived, or discarded.

[11]see https://martinfowler.com/bliki/Infodeck.html

4.3.2.5 How to effectively and efficiently provide documentation in continuous software design specification?

To remedy some of the aforementioned problems, the focus group participants discussed how the effort for creating documentation artifacts in CONTINUOUS SOFTWARE DESIGN SPECIFICATION can be minimized, while still providing the necessary information to the stakeholders. In other words, how can documentation be created effectively and efficient? At first, the idea is intriguing to simply reuse specification artifacts as documentation. However, as mentioned above, this comes with the downside that the documentation may contain gaps and inconsistencies. One idea to tackle this problem was to apply practices known from source code refactoring also to the documentation process. In software development, refactoring refers to the process of restructuring existing source code to make it more effective, or to make it more maintainable without changing the behavior of the software. Likewise, refactoring specification into documentation would apply to improving the expressiveness and suitability of existing specifications so that they can be used as documentation. Refactoring specifications leads to documentation without changing the design of the software. We discussed the following (initial and incomplete) specification refactoring:

- **Split to stick to single responsibility.** The single responsibility principle is taken over from agile software development [72]. Interpreted in the context of documentation, the principle states that every specification or documentation item should cover one specific aspect or part of the software that should entirely be encapsulated by this documentation item. Groups of stakeholder concerns addressed by viewpoints can be used to source these responsibilities.
- **Apply open/closed principle.** When refactoring specifications, make sure they are open for extension, but closed for modifications. This principle is adapted from the corresponding principle used by the agile software development community [72]. Applied to specifications, you need to make sure that adding additional information does not require (major) rework of the existing documentation. Semantic versioning[12] should be applied to specifications and documentation items just like for code.
- **Remove repetition.** Do not repeat yourself is a lean and agile tenet, so one should not provide the same piece of information in different documentation artifacts. Repetition results in increased efforts and higher risk of inconsistency when information is changed. The "definition of done" for specifications and documentation should include a check whether the same things have already been said elsewhere.
- **Replace specification by realization.** Related to the previous item, once a specified software artifact was realized, refer to the realization instead of providing the same information in a different way. To give examples, instead of describing object interaction using a UML-sequence diagram, let the code speak for itself. Contemporary Integrated Development Environment (IDE)s have become so powerful that exposing source code in the IDE is preferred by many developers over reading sequence diagrams.
- **Document general structure instead of concrete realization.** Instead of repeating information that is also provided by the source code itself, describe the general structure of a software artifact or pick one example and explain how the example can be adapted to other cases, as well.
- **Throw specifications away.** Some specifications have fully served their purpose when a software artifact was realized. These specifications can be thrown away in the sense that they can be deleted and only kept in the history of the knowledge management tool (e.g. Confluence as mentioned above).
- **Provide yellow pages.** Instead of striving for one large self-contained and complete document, split up documentation items into smaller coherent chunks and provide overview pages with links to these smaller chunks. A documentation chunk can be text, source code, a diagram, a model, a whiteboard sketch or anything else that has value as documentation.

We also discussed when and how this process could take place in CONTINUOUS SOFTWARE DESIGN SPECIFICATION. Most participants said that the end of an iteration would be the most suitable point in time. Ideally, documentation refactoring is done in pairs to decide when a documentation artifact is clear enough for the indented audience. Reading the text out loud during the pair documentation refactoring session can be an intense and highly productive experience. When preparing documentation, one should make good use of the features provided by the typical tool suites used in CONTINUOUS SOFTWARE DESIGN SPECIFICATION, e.g., for cross-referencing between wiki pages, tasks, requirements, code, and quality metrics.

[12] see http://semver.org/

4.4 Future Work

We plan to perform further research on how architecture specification and documentation can be better aligned with CONTINUOUS SOFTWARE DESIGN SPECIFICATION practices and tools. As part of this research, we will investigate further into the purposes, efforts, and differences between specifications and documentation, including the preservation of rationale that went into architectural decisions made. In particular, research on DevOps and continuous delivery in relation to architecture is interesting here, as well as ensuring that architecture models are closer aligned to other software engineering artifacts like source code and configuration files, for instance. The research results will be used to develop a lightweight architecture framework that embraces the principles of continuous software development.

4.5 Acknowledgments

We would like to thank all participants of the focus group on software specification in continuous software development, which took part at EuroPLoP 2017.

4.A. Appendices

4.A. A. POSTER

Specification in Continuous Software Development

PRINCIPLES OF CONTINUOUS SOFTWARE DEVELOPMENT

Principle	Lean	Agile	DevOps
1. Efficiency, effectiveness	1. Eliminating waste	1. Customer satisfaction 7. Working software is progress 8. Sustainable pace 10. Simplify	3. Measurement
2. Learning	2. Amplify Learning	12. Regular adjustments	4. Sharing
3. Flexibility	3. Decide as late as possible	2. Welcome changes	1. Culture
4. Time to market	4. Deliver as fast as possible	3. Frequent releases	2. Automation
5. Trust, attitude	5. Empower the team	5. Trusted individuals	7. Structures and standards
6. Improvement, competences	6. Build integrity in	9. Technical excellence 11. Self-organizing teams	6. Quality assurance
7. Big picture	7. See the whole	6. Face-to-face conversation	5. Services
8. Involvement	-	4. Collaborate with business people	7. Structures and standards

The context describes a situation where the problem occurs and the solution is applied, i.e. specification in continuous software development.

CONTEXT

1. You have deliberately chosen to apply the principles of **continuous** software development;
2. Your **team has worked together** on multiple software products;
3. The people in your **team know each other well** and have an established communication culture;
4. The team members are also **knowledgeable** about the technological domain, in which the software product to be developed resides.

The problem is how to provide just-enough adequate specifications.

PROBLEM

- Developers strive for **omitting artifacts** that are not immediately required for building a high-quality software product.
- Just another view on a software product that is already specified by the source-code is considered as **wasted effort**.
- Issues when specifications are not sufficiently defined:
  - Reasoning about architectural problems;
  - Supporting planning activities;
  - Defining interfaces between team members.

Fo... when deciding on a solution for the problem at hand. Forces are often conflicting.

FORCES

1. Shaping thoughts
   a. The process of specifying contributes to a better understanding of the problem and envisioned solution of an application.
2. Progressive insight
   a. During a software development process, developers gain new insights that they need to consider in the implementation
3. Specification gaps require assumptions to be made
   a. Silent assumptions bare the risk that individual team members make decisions that interfere with or even contradict each other.
4. Hidden disagreements
   a. Developers discuss a problem or an envisioned solution and actually talk across purposes without realizing the agreements.
5. Overestimated competences and underestimated complexity
   a. People tend to overestimate their own competences and skills
6. Time to market
7. On-boarding of new team members
   a. Application-specific design knowledge needs to be transferred to the new people in charge
8. Explored design space
   a. Preserved knowledge from the past like templates, frameworks and libraries.

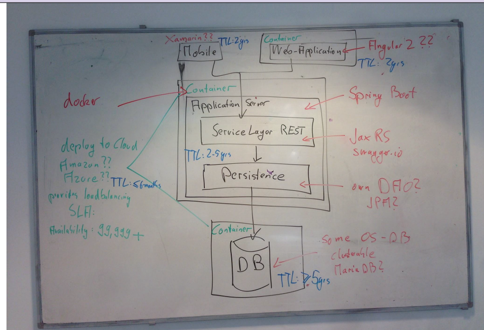
The solution describes a rule how conflicts within the forces can be resolved.

SOLUTION

1. Specifications at the **start** of an iteration
   a. A list of **requirements** to be addressed in the iteration.
   b. An architectural **vision**.
   c. A description of the technological **ecosystem** in which the software will be developed.
   d. Information about the most important architectural **concerns** (i.e. quality attribute requirements and business drivers), which determine the priorities of decisions to be made.
2. Specifications **during** an iteration
   a. Specifications to *continue* development during an iteration cycle should **codify agreements** made between individual team members or between multiple teams working on the same larger application.
3. Refactored specifications at the **end** of an iteration
   a. Revisit specifications created and updated (**refactored**) during the iteration and decide explicitly on items relevant for the next iterations;
   b. A description of (automated) steps to **test, deployment and operations**.

Uwe van Heesch ● Theo Theunissen ● July 2017

WHITEBOARD SKETCH



SPECIFICATIONS

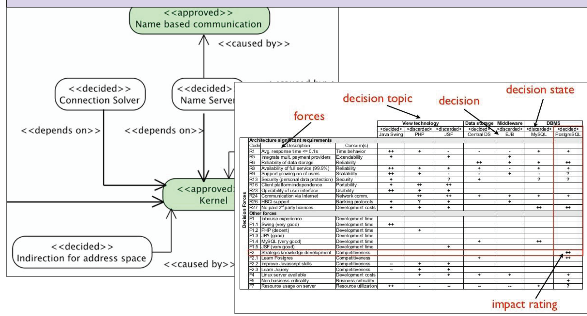
```
Scenario: travel with two children
  Given I am at www.ns.nl
  And I clicked accept in cookie popup
  And I choose to travel today
  And I fill "Utrecht" as the station Vanaf
  And I fill "Amsterdam" as the station Naar
  And I select "1e" as the Klasse
  And I select "1" as the number of Reizigers
  And I fill initials "J.J.G.W.M." for traveller "1"
  And I fill name "Bakker"
  And I fill birth date "2
  And I have clicked on In
  And I have clicked on Ra
  And I have clicked on Di
  And I fill to travel tod
  And I select "2" as the
  And I fill "J." as the f
  And I fill "Bakker" as t
  And I fill "22-11-2010"
  And I fill "R." as the s
  And I fill "Bakker" as t
  And I fill "02-03-2009"
  When I have clicked on K
  Then the total price is
```

REST specification in YAML

Documentation and Test console

Uber API

DECISIONS



DISCUSSION ITEMS

1. What is the difference between specification and documentation? What purposes do they serve and for whom?
2. What is the role of architecture specification in continuous software development?
3. Should software specifications be organized around self-contained documents?
4. What is the role of models in this context?
5. What alternate forms of organizing specification items could be a better fit for continuous software development?
6. What is the life cycle and scope of the different types of specifications and how does this life cycle relate to the agile lifecycle, for instance?
7. Refactoring has become a common technique for improving the structure and quality of source-code. How can similar techniques be used for specifications?
8. What is the role of (architectural) design decisions in this context? How to share/document/update.. them?
9. How to efficiently preserve (architectural) knowledge and skills a development team gained throughout multiple iterations/projects? What difference exists between generic and application-specific knowledge and skills?
10. How can agile practices (e.g. sprint retrospectives) be leveraged in this context?
11. How can information needs by external reviewers and auditors be satisfied?
12. What's the impact of business and technical domain specifics in this context?

europlop

Hogeschool van Arnhem en Nijmegen
HAN University of Applied Sciences

Chapter 5

A Mapping Study on Documentation in Continuous Software Development[1]

Abstract Context: With an increase in Agile, Lean, and DevOps software methodologies over the last years (collectively referred to as Continuous Software Development (CSD)), we have observed that documentation is often poor.
Objective: This work aims at collecting studies on documentation challenges, documentation practices, and tools that can support documentation in CSD.
Method: A systematic mapping study was conducted to identify and analyze research on documentation in CSD, covering publications between 2001 and 2019.
Results: A total of 63 studies were selected. We found 40 studies related to documentation practices and challenges, and 23 studies related to tools used in CSD. The challenges include: informal documentation is hard to understand, documentation is considered as waste, productivity is measured by working software only, documentation is out-of-sync with the software and there is a short-term focus. The practices include: non-written and informal communication, the usage of development artifacts for documentation, and the use of architecture frameworks. We also made an inventory of numerous tools that can be used for documentation purposes in CSD. Overall, we recommend the usage of executable documentation, modern tools and technologies to retrieve information and transform it into documentation, and the practice of minimal documentation upfront combined with detailed design for knowledge transfer afterwards.
Conclusion: It is of paramount importance to increase the quantity and quality of documentation in CSD. While this remains challenging, practitioners will benefit from applying the identified practices and tools in order to mitigate the stated challenges.

Keywords Agile, DevOps, Documentation, Continuous Software Development, Lean, Systematic mapping studies, Systematic reviews

5.1 Introduction

In recent years, we have seen an increase in the adoption of Lean and Agile software development, as well as DevOps. In our previous work [30], [S31], [S32], we have introduced the term *Continuous Software Development* (CSD) as an umbrella term to collectively refer to such development processes and other processes that share

[1]This work was originally published as:

T. Theunissen, U. van Heesch, and P. Avgeriou, "A Mapping Study on Documentation in Continuous Software Development," *Information and Software Technology*, vol. 142, p. 106733, 2022, doi: 10.1016/j.infsof.2021.106733.

the following characteristics:

1. it covers the values, principles and practices from Agile ([1]), Lean ([2]) and DevOps.
2. it embraces activities from the whole life cycle of a software product, from concept to end-of-life. In addition to Agile and Lean software development, it includes maintenance activities. In addition to DevOps, it includes continuous architecting activities ([3]).
3. it considers the continuously changing state of the software product and progress, such as progressive insights (e.g. regarding process, design, implementation), changes in contextual factors, new features, bug fixes, or other unforeseen factors.
4. it distributes information about software development across multiple tools, because of demands for fast time-to-market, as well as the need for a software development ecosystem for automated tests, deployment, and monitoring. Thus, no central repository for all information is available.

One of the main challenges in CSD is that documentation is poor [1], [2], [73]. This challenge hinders knowledge transfer [S74], has a bad impact on maintenance [S75] and introduces a steep learning curve [S76] for new team members. We elaborate further on these consequences. First, knowledge transfer is hindered when knowledge about the software product, such as decisions, bugs, context, and practices, remains implicit in the minds of developers and is only informally written in whiteboard sketches. As a result, knowledge walks literally out the door at the end of a daily stand-up or even leaves the company (many companies have high staff turn-over) [77]–[79]. Second, it is hard to act when bugs show up, new features or non-functional requirements arise. Developers are forced to make assumptions about decisions, interfaces, or priorities; such assumptions are often wrong [80]–[82]. Third, the system is hard to understand for the different stakeholders, including developers. Especially, when the team scales up, or team members switch to other projects, newcomers go through numerous trial-and-error attempts before they can contribute well [83]–[85], [S86].

There is plenty of information in the different tools that are used, but that is mostly related to implementation, deployment and operations. The following, exclusively distinctive types of information are often lacking, incomplete, out-of-date, or of low quality [21]:

1. *Stakeholders and their concerns.* This is key in prioritizing requirements and mitigating risks. A stakeholder is anyone who has an effect on the system or is affected by the system [87], [88].
2. *Risks.* Risks can endanger the project [63], and manifest as incomplete information, lack of information, or factors that are out of control of the development team.
3. *Assumptions and constraints.* Both delimit the solution space, but are very often tacit or implicit [S89].
4. *Context and environment.* This includes anything that has an effect on the system but is not included in the primary goals, such as legal[2] and environmental issues[3] [90].
5. *Design decisions and their rationale.* The rationale typically concerns trade-offs between qualities, business factors, in-house expertise etc. [S61], [69].
6. *Design and/or architecture specifications.* Even if design specifications are created, they are typically not updated according to changes in requirements and context, and thus become out-of-sync with the actual code [91].

As a first step in addressing the problem of poor documentation in CSD, we decided to look into the current state of practice as reported in scientific literature. Specifically, we conducted a systematic mapping study on identifying the challenges of documentation in CSD as well as the practices and tools that can potentially support documentation. We selected to study these aspects in order to shed light into both the problem (documentation challenges) and the solution (practices and tools); we note that practices and tools are the primary means for architecture documentation [92]. Our aim is to shed light on what is currently on offer for documentation purposes in a CSD context, as well as what is still lacking.

Our results indicate that documentation is considered waste in Lean development when it does not contribute to the end product. Consequently, developers tend to minimize documentation or leave it out. Furthermore, documentation is often out-of-sync with the software, irrespective of whether documentation is within the source code or documented in wiki-like systems. Moreover, the focus is only short-term: knowledge about design decisions, practices, and lessons learned are within a team, primarily when the team is gathered in a single geographical location. The practices we discovered are that written documentation is left out, and communication is informal, while development artifacts are used as a specification. Finally, the use of architecture frameworks can also support sound documentation.

We decided to conduct a SMS instead of a systematic literature review (SLR). SMS are typically used for newer research topics where there are few or no secondary studies and the main objective is to classify

[2]For instance privacy as defined in the General Data Protection Regulation (GDPR).

[3]E.g. low CPU consumption.

and conduct a thematic analysis of literature [93], [94]. Further motivation for the use of SMS versus SLR is provided in the beginning of Section 5.

5.1.1 Research Questions

We formulate the goal of the study using the format of the Goal-Question-Metric (GQM) approach [95]: **Analyze** literature **for the purpose of** exploration, characterization and analysis **with respect to** documentation challenges, practices, and tools **from the point of view of** researchers and industry practitioners **in the context of** Continuous Software Development. Based on the aforementioned goal, we have set the following research questions:

RQ1 What are the documentation challenges and specific practices in CSD?
We already know of several challenges in CSD. We have established that poor documentation hinders knowledge transfer [S74], which, in turn, has a bad impact on maintenance [S75] and introduces a steep learning [S76] curve for new team members. Furthermore, documentation seems to have a lower value in CSD, than in traditional software development processes such as Rational Unified Process (RUP) [96]. For example, the Agile Manifesto explicitly values working code over written documentation; face-to-face communication is considered the most effective way of conveying information [1]. In Lean software development, documentation is often considered waste, as it does not directly contribute to customer satisfaction [2]. In DevOps, infrastructure is key to fast deployment and information is represented as code [73], rather than written documentation. With this research question, we aim at understanding in more depth such challenges that work against documentation in CSD. We also strive to uncover potential practices that result in successful documentation in the context of CSD.

RQ2 Which tools from the Continuous Software Development ecosystem can be used for documentation purposes?
CSD relies heavily on tooling in order to achieve faster deployment, continuous testing, and monitoring quality [S97]. These tools contain much information about source code and configuration (e.g. git), test cases (e.g. Cucumber), deployment (e.g. Docker or Jenkins) and quality (e.g. SonarQube). This information would typically also be documented in software design description documents. With this research question, we want to understand which tools are used in CSD and how they could additionally be exploited for documentation purposes.

5.1.2 Related Secondary Studies

There are several secondary studies on the topics of Lean, Agile and DevOps. In the following, we describe those studies that discuss *documentation* in Lean, Agile and DevOps. We also present the reason why they are related and which gap our study attempts to address. These five studies address issues, describe industry practices, and propose and explore processes, tools and methods.

Rodríguez, Haghighatkhah, Lwakatare, *et al.* [98] analyze the body of knowledge in Continuous Deployment [98]. They give an overview of concepts and typical characteristics of continuous deployment, such as fast and frequent releases, and continuous automated testing. The authors emphasize the importance of tools for supporting continuous integration and continuous delivery, but they do not address the relation between tooling and documentation.

Diebold and Dahlem [99] looked into agile practices in the industry under different circumstances, such as different project types, domains, or processes [99]. They found that agile practices appear in most projects across several industry domains. Such agile practices are used in methods like Scrum, Kanban, and eXtreme Programming (XP). The study focuses specifically on the development activities that lead to the first major release, whereas maintenance concerns are not particularly taken into consideration. The study does not cover documentation in agile projects.

In two different secondary studies on requirements engineering in agile software development, Heikkilä, Damian, Lassenius, *et al.* [100] and Curcio, Navarro, Malucelli, *et al.* [101] independently found that there is no clear line on how requirements engineering activities should be performed in agile processes; the overall understanding of requirements engineering in agile software development is still rather immature. Both studies report that an agile development team usually comprises highly skilled and experienced developers who act on their knowledge and skills; this knowledge and the thought process of developers is usually not written down in documents, i.e. agile teams rely mainly on tacit knowledge. Furthermore, these highly skilled professionals are often required for other jobs and frequently switch teams. New team members, who might be less qualified and experienced, do not know the decisions and actions taken. Heikkilä, Damian, Lassenius, *et al.* [100] suggest

that knowledge should be written down for new team members [100]. Neither the study of Heikkilä, Damian, Lassenius, *et al.* [100], nor the one of Curcio, Navarro, Malucelli, *et al.* [101] discussed documentation practices, and tools.

Shafiq and Waheed [102] found that agile development teams often make use of predefined document templates as a means for efficient standardization [102]. For instance, Feature-Driven Development (FDD) uses templates for use cases and functional requirements, Scrum uses user stories (as <role>, I want <objective> because of <rationale>). Generally, agile teams avoid recording long, complex, strictly-defined or rigid pieces of information in textual documents.

In summary, documentation concerns in CSD are gaining attention within the research community. However, there is currently no consensus on concrete documentation practices. There is no practice or documented tooling that can be used for documentation purposes; instead, information is distributed across software development tools.

5.1.3 Paper Structure and Reference Styles

The remainder of this document is structured as follows: in Section 5, we present the study design. Section 5 provides demographic information about the selected primary studies. In Section 5.11, we discuss the results and provide our own interpretation, as well as implications for researchers and practitioners. Finally, we discuss potential threats to the validity of this work in Section 5.11.

We use two styles of references in this study. One style refers to the primary studies that are analyzed to answer the research questions. These references are denoted with an S (for study) and a number within square brackets, e.g. [S123] refers to study 123 that is shown in 10. The other style of reference is without the ‘S’; it refers to papers that do not belong to our set of primary studies but are used for other purposes (for instance in the Related Secondary Studies section) and can be found in the References.

5.2 Study Design

As a method to conduct this literature study, we considered a SLR, as well as a SMS. Table 5.1 is adapted from Kitchenham, Budgen, and Brereton [93] and compares typical characteristics of the two methods.

Characteristic	SLR	SMS
Goals	Identification of best practices	Classification and thematic analysis of literature
Research Questions	Specific - related to outcomes of empirical studies	Generic - related to research trends
Search process	Based on research questions	Defined by topic area
Scope	Focused - outcomes of empirical studies	Broad - includes non-empirical studies
Search strategy requirements	Exhaustive - all relevant studies should be found	Less stringent
Quality evaluation	Important. Results must be based on best-quality evidence	Not essential. Non-empirical studies may make quality evaluation hard
Results	Using outcomes of primary studies to answer specific research questions	Categorization of papers into dimensions

Table 5.1: Comparison of typical characteristics in literature research methods

Using these seven characteristics, we justify why we used the systematic mapping study as follows:

1. **Goals.** We want to present a broad overview of literature and to categorize this literature in dimensions.
2. **Research Questions.** We address broader research questions regarding the trends in documentation challenges, practices, and tools in CSD.
3. **Search Process.** We are looking into a specific topic area: documentation in CSD.
4. **Scope.** We focus on both empirical and non-empirical studies. The topics of Agile, Lean and DevOps are very practitioner-oriented, thus we expect that, at least part of literature is not empirical.
5. **Search strategy requirements.** We are looking at trends, so we can afford to be less stringent.
6. **Quality Evaluation.** The combination of non-empirical and empirical studies makes it complicated to evaluate the quality of primary studies.
7. **Results.** We aim at classifying papers into dimensions.

Based on these reasons, we chose a systematic mapping study over a systematic literature review. We follow the guidelines of Petersen, Feldt, Mujtaba, *et al.* [103] for systematic mapping studies [103]. Figure 5.1 depicts the steps of the study as well as the steps of the study protocol. Arrows pointing in both directions indicate

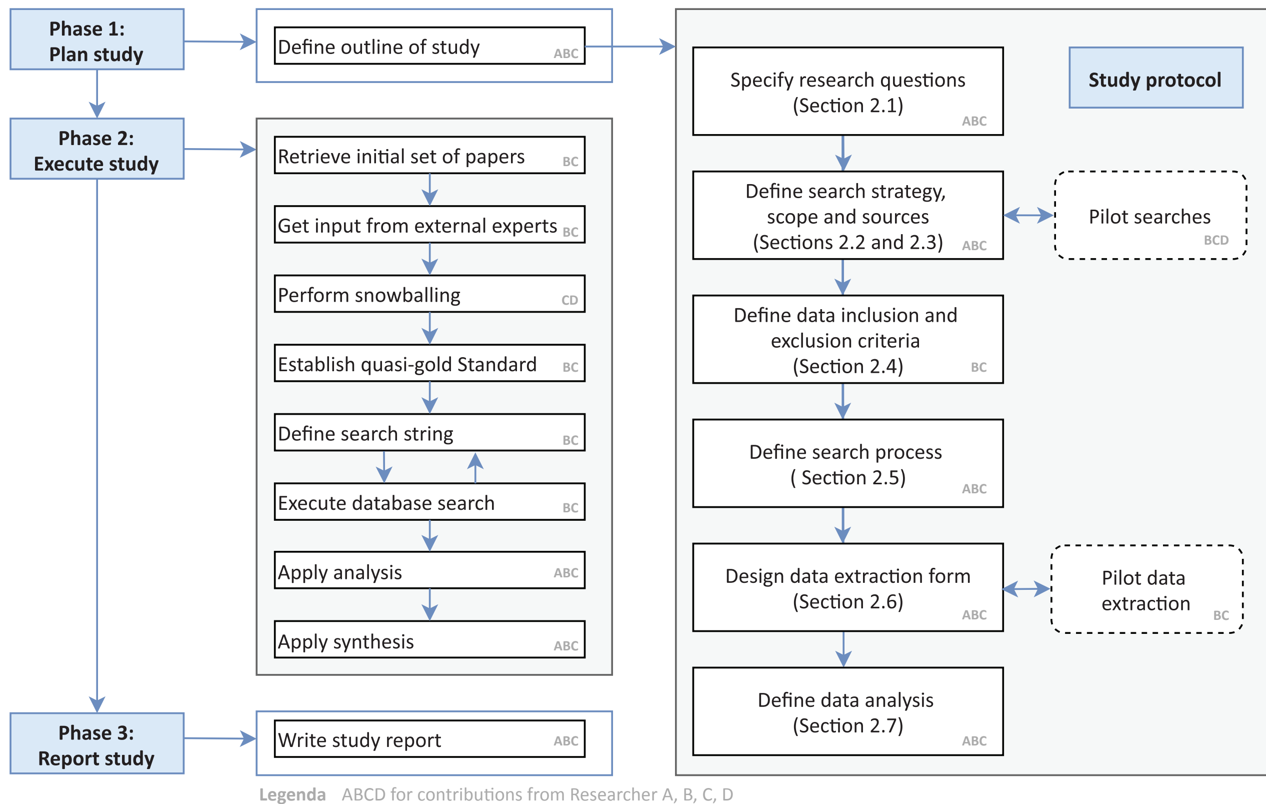


Figure 5.1: The study process and the protocol

that steps were performed iteratively. In the following sections, we briefly describe each of the steps of the study protocol (right part); the steps of "Phase 2: Execute study"(left part) are elaborated in Section 5.

Figure 5.1 shows the contributions for all team members for each process step. The team comprises four researchers (labeled A, B, C, and D), varying in seniority. Two researchers selected studies, read the title, keywords, abstract and full paper. This resulted in a raw result set with candidate studies. Two other researchers read only the title, keywords and abstracts of the studies in the raw result. Studies that did not contribute to answering the research questions were rejected from the final result set. Finally, all team members read papers from the final result set. In Figure 5.4, we made a distinction between raw results with candidate studies and final result sets with studies that contribute to answering research questions. Thus, it shows results per step and studies that were read concerning titles, keywords, abstracts, and full papers. By making a distinction between raw results and final results, we established a process for reaching consensus.

5.2.1. Search Strategy

The search process combines a manual process with automated search. A manual search process typically has a higher accuracy than automated search, because it focuses on targeted venues, but it also has a risk of bias, because of the researcher's personal preferences. Additionally, it is more time-consuming. Other criteria such as transparency and reproducibility are hard to achieve with a manual search, even if all quality and evaluation criteria are explicitly defined [104], [105]. Furthermore, automated search is typically more comprehensive than manual searches [104], [105]. We therefore decided to apply a combination of both methods. The manual search process, as well as the automated search will be further elaborated in Section 5.

5.2.2 Scope of Search and Sources Searched

The scope for this study is limited by the following criteria:

1. The study is published between January 2001 (i.e. the publication of the Agile Manifesto ([1]) and February 2019 when the writing of this report started;
2. The study can be found in scientific databases in the field of software engineering, that include journals, conference papers, and workshop papers; the following sources were used: ACM, IEEE, ScienceDirect,

SpringerLink and WebOfScience. These databases are quite commonly used in secondary studies in Software Engineering [106].

5.2.3 Inclusion and Exclusion Criteria

Inclusion and exclusion criteria help to make a transparent and reproducible selection of papers in the mapping study. Papers are included if they meet all of the inclusion criteria and excluded if they meet any of the exclusion criteria. The criteria are exhibited in Tables 5.2 and 5.3.

ID	Inclusion criteria
I1	PDF or full text must be available
I2	Domain or discipline must be software engineering
I3	Study must be written in English
I4	Study must be peer reviewed
I5	The search terms must appear in title, keywords or abstract

Table 5.2: Inclusion criteria

ID	Exclusion criteria
E1	Study is a duplicate of another study in scope

Table 5.3: Exclusion criteria

5.2.4 Search Process

The search process follows the steps in the *execution phase* (see Figure 5.1, left part). The study was conducted by four researchers. The data collection was done by the corresponding author with the assistance of a master student; the analysis and interpretation was performed by all authors. We began the process with the snowballing technique by following Wohlin [107] guidelines (see Figure 5.2) [107]. Specifically, we formed an initial set of papers on subjects, topics, and authors we found relevant for this SMS. Additionally, we asked subject-matter experts from academia and industry to come up with papers they deem primarily relevant for this study (see appendix 5.11). With the resulting set of papers, we conducted the snowballing technique until no more relevant publications could be found.

The resulting set of papers was used to define a Quasi-Gold Standard (QGS), which is a "*well-known*" set of papers that are relevant to evaluate the results and to establish a search string for an automated search ([108]). The search string was based on the QGS. The performance of the search string was performed by comparing the results of the automated search with the QGS: all papers from the QGS were returned from the automated search.

Subsequently, we defined the search string, based on the words from the title, keywords and abstract from the papers in the QGS. We used the n-gram procedure to assist us in establishing the search string [109]. Specifically, we first removed 1,000 common English stop words[4]. Next, from the remaining words we took sequences of words to cover the domain (CSD) and the research questions. A manual step was required to adjust the search string to make it more efficient by removing unnecessary terms. Especially for RQ2, we added a wildcard to leave the single "document" out, as searching for "document" resulted in too many irrelevant hits. The resulting search string we used for the automated search is:

```
-- domain
(  lean
OR agile
OR DevOps
OR "continuous software"
OR scrum
OR "extreme programming"
```

[4]https://gist.github.com/deekayen/4148741

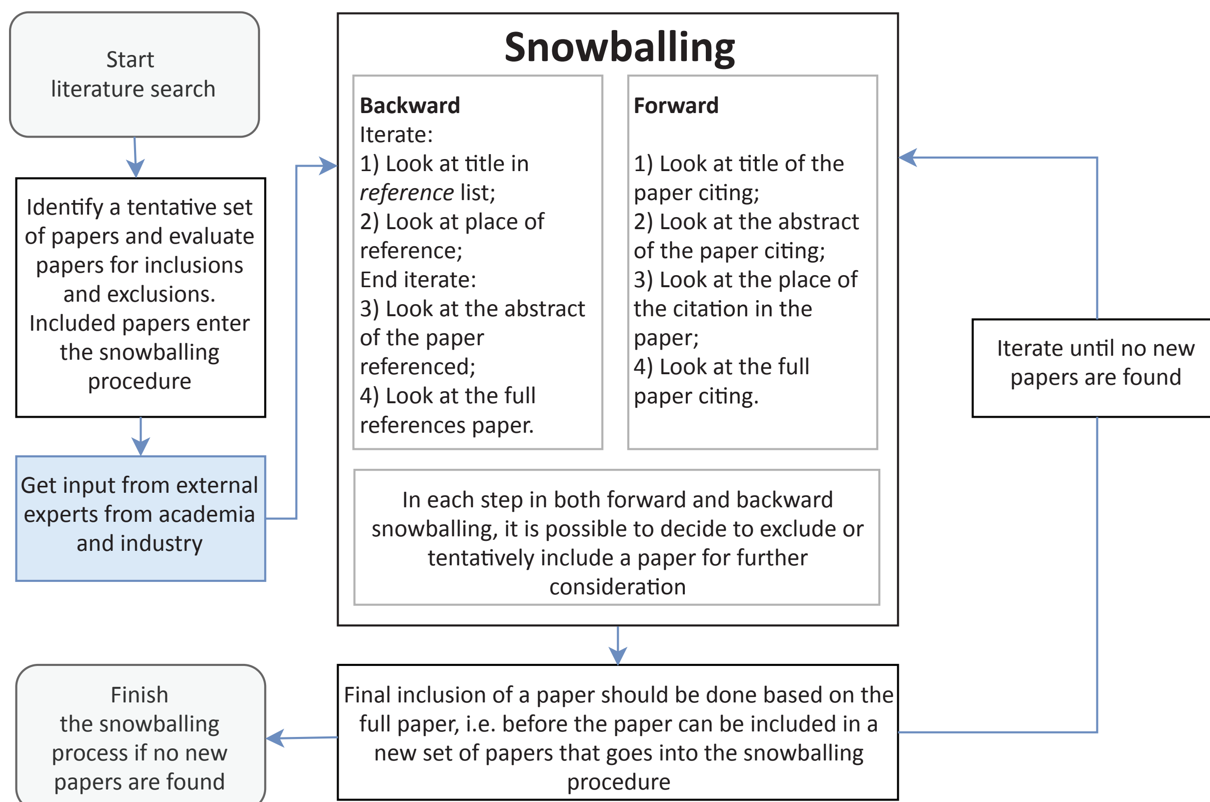


Figure 5.2: The snowballing search process (adopted from Wohlin [107])

```
)
-- RQ1
AND (
documenti*
OR documenta*
)
-- RQ2
AND (tool*)
)
```

With the search string defined, the execution of the database search was performed. For this, we scraped the meta-data from the online libraries to store it locally in our database. The reason for local storage is to compare studies equally. In the first place, the online search engines all do have a different query language which look similar when it comes to syntax, but the one online library is more precise in targeting than the other online library, especially the discipline (e.g. SpringerLink) or domain (e.g. ACM). Second, the online libraries do not use the same data model for the bibliographical data, for instance, the meta-data has a different format (BibTeX, RIS). Another difference is the type of the fields, such as the fields for "authors", "titles" and "keywords" might either be a string or a list. The third reason is to be able to add tags, labels and comments for answering research questions.

5.2.5 Data Extraction

For each study, the information shown in table 5.4a was collected. The attributes for the *title* (F1), *keywords* (F2) and *abstract* (F3) where used for snowballing and calibration of the search string. We used Mendeley[5]

[5]https://www.mendeley.com

ID	Attribute	Usage
F1	Title	snowballing, search string calibration, synthesis
F2	Keywords	snowballing, search string calibration
F3	Abstract	snowballing, search string calibration
F4	Full text	exploration
F5	Publishers database	inclusion / exclusion
F6	Year	inclusion / exclusion
F7	Documentation, documenting, architecture framework	RQ1
F8	Tools, tooling	RQ2

(a) Data Extraction Form

for storing and tagging the papers during the initial phases. With the tags it was easy to select the papers. We commented the papers with keywords and comments for relevance, as well as terms that can be used for the search string. The suggestions for studies from external experts were also stored in Mendeley and tagged accordingly. During snowballing and establishing the Quasi-Gold Standard, Mendeley was used to store, comment the papers and add keywords. With the automated search, the results were too many to store in Mendeley, thus we used a database to store them. The database was structured according to the basic BibTex bibliographic references[6], supplemented with extra fields for additional keywords, categories, and concepts.

The attribute *full text* (F4) was used in exploring the research area in the pilot search. The attribute values for *publishers database* (F5) and *year* (F6) were required by the inclusion and exclusion criteria. Attributes F7 and F8 were used for answering the two research questions.

5.2.6 Data Analysis

For the analysis of quantitative data, we used descriptive statistics. For the analysis of qualitative data, we adapted the approach of Schwandt [110], as depicted in Figure 5.3. As supporting tooling, we used Atlas.ti[7]. We started with the studies from the final result set. Next, we read the studies and marked text when it answered or contributed to a research question. This resulted in marked text. In the second step, we coded the marked text with keywords that characterize the fragment. The keywords could be individual words from the marked text but also other words that are typical for the marked text. We also used an online thesaurus and used Google to come up with additional or alternative keywords. The result of the coding step is a list of keywords. The third activity was grouping keywords into categories. Categories are a higher-order abstraction of the keywords. The result of the grouping of keywords is a list of categories. The fourth activity concerns identifying relations between categories. The relations denote connections among categories. The types of relations are derived from UML: activity edges, associations, dependencies, generalizations, realizations, and transitions. We kept the number of relations to a minimum to have clear distinctions between the resulting concepts. The activities for keywords, categories, relations, and concepts were iterated until no more refinement was possible. The last activity was the mapping of the concepts on the systematic map (see Figure 5.7).

5.3 Results

This section describes the results of the mapping study, according to the guidelines of Petersen, Feldt, Mujtaba, *et al.* [103]. First, we show the demographic data of the identified studies (Section 5). Then we classify the studies according to our research questions using a facet map (the systematic map) in Section 5. Finally, we discuss the results in the context of each individual research question: RQ1 in Section 5 and RQ2 in Section 5.

5.3.1 Demographic Data

As described in 5, we used a two-fold search strategy comprising 1) a purely manual search based on input from subject-matter experts and snowballing and 2) an automated search. The results from both search types were merged, resulting in 58 unique papers that were used for answering our research questions. Figure 5.4 illustrates this process again together with the numbers of papers resulting from each individual step. The initial set of four papers was relevant content-wise, but did not pass our inclusion criteria, because they are not

[6]https://en.wikibooks.org/wiki/LaTeX/Bibliography_Management
[7]https://atlasti.com/

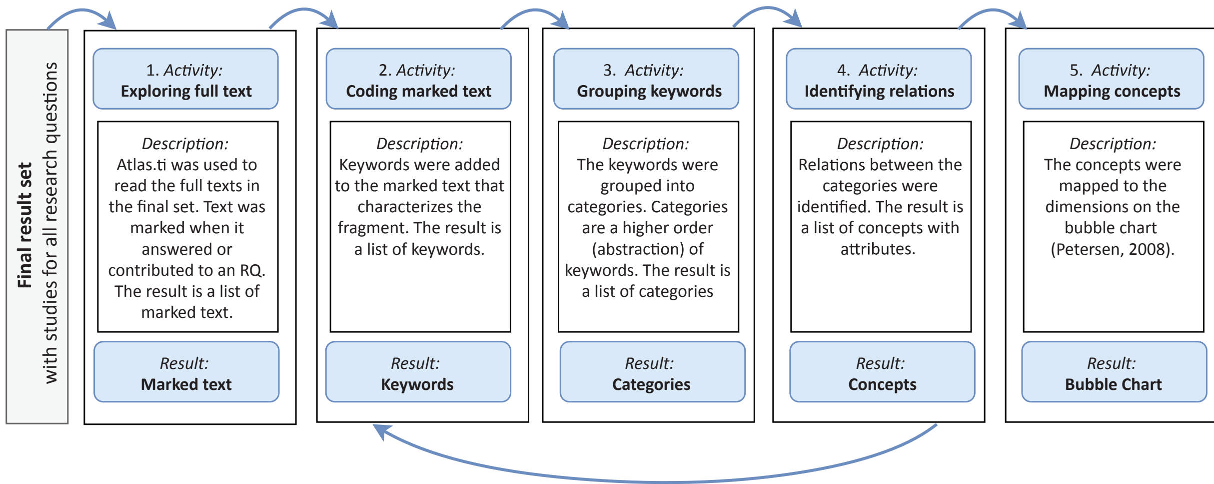


Figure 5.3: Qualitative classification process on the final set (Schwandt [110])

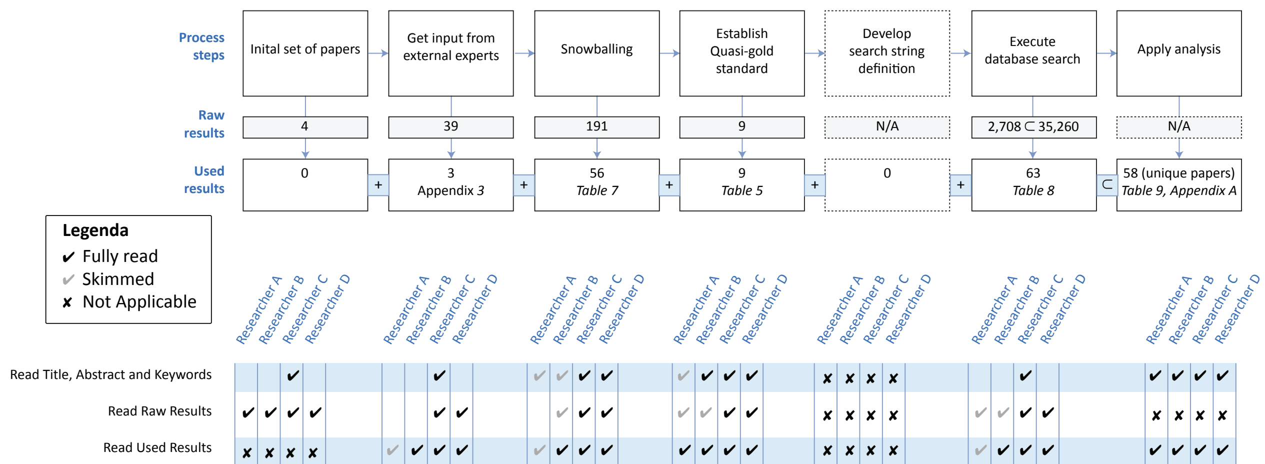


Figure 5.4: Overview of search results per step.

scientific studies. Nevertheless, they served as a basis for the snowballing procedure, together with the input from the external experts, who suggested 39 articles in total (see Appendix 5.11), out of which three papers matched our criteria. The snowballing procedure delivered 92 studies. We studied these articles to select a set of nine studies (shown in Table 5.5) that we consider a Quasi-Gold Standard.

The QGS was used to validate the search string for the automated search, i.e. we tweaked the search string iteratively until all studies in the QGS ended up in the results of the search. Table 5.6 shows the search string, its relation to the research questions, and the number of hits in abstracts, keywords, or title, respectively. The column *Intersection* shows the number of papers in which the search term was found in all three parts of the studies. In total, we ended up with 58 unique papers that relate to our research questions (i.e. 40 for RQ1 plus 23 for RQ2 makes a total of 63 non-unique papers).

Table 5.7 shows the papers identified during the manual search, Table 5.8 shows the automated search results. The union resulted in 58 unique studies (see Table 5.9) that were analyzed in the next step.

The distribution of studies according to publication types is displayed in Figure 5.5. About 80% studies are from conferences.

Figure 5.6 plots the publication years of all identified studies. Clearly, the topic is increasingly gaining attention in the research community. decided to look into the current state of practice

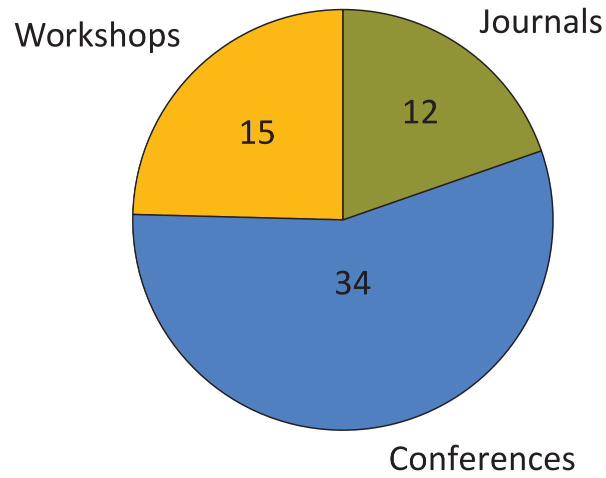


Figure 5.5: Distribution per year

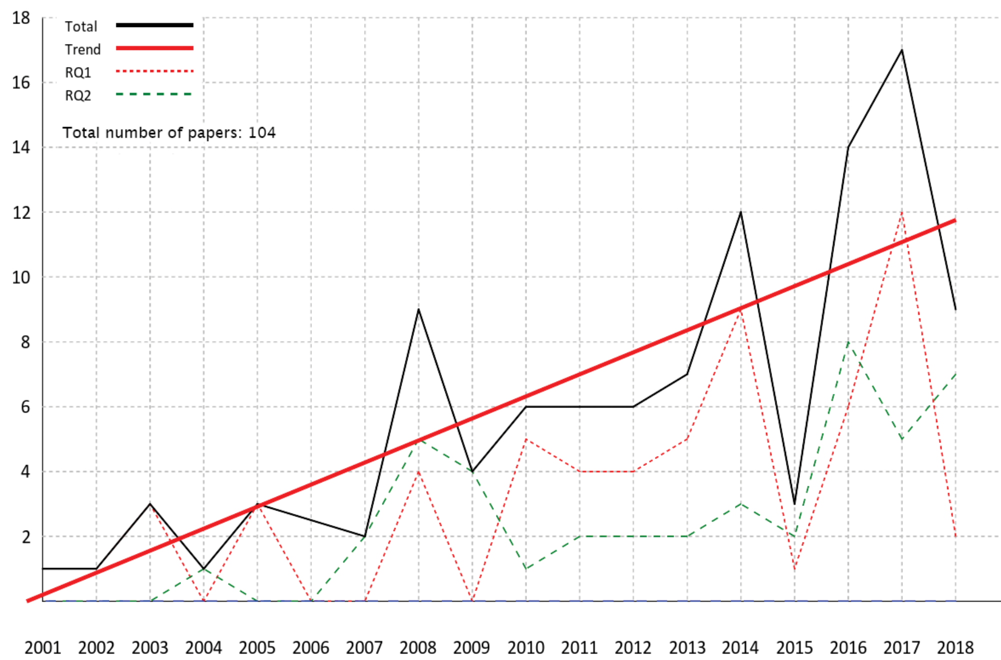


Figure 5.6: Distribution per year

ID	Publication
[S111]	*A Study of Documentation in Agile Software Projects* - Voigt, Stefan; von Garrel, Jorg; Muller, Julia; Wirth, Dominic
[S112]	*A study of the documentation essential to software maintenance* - de Souza, Sergio Cozzetti B; Anquetil, Nicolas; de Oliveira, Káthia M
[113]	*Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects*; A. Rüping
[S32]	*Software Specification and Documentation in Continuous Software Development: A Focus Group Report* - Van Heesch, U.; Theunissen, T.; Zimmermann, O.; Zdun, U.
[S31]	*Specification in Continuous Software Development* - Theunissen, T., Van Heesch, U.
[S114]	*SprintDoc: Concept for an agile documentation tool* - Voigt, Stefan; Huttemann, Detlef; Gohr, Andreas
[S115]	*Supporting agile software development through active documentation* - Rubin, Eran; Rubin, Hillel
[S116]	*Towards the essentials of architecture documentation for avoiding architecture erosion* - Gerdes, Sebastian; Jasser, Stefanie; Riebisch, Matthias; Schröder, Sandra; Soliman, Mohamed; Stehle, Tilmann
[S117]	*Using design rationales for agile documentation* - Sauer, T

Table 5.5: The Quasi-Gold Standard

RQ	Query	Abstract	Keywords	Title	Union	Intersection
(BASE QUERY)	`(lean OR agile OR DevOps OR 'continuous software' OR scrum OR 'extreme programming')`	8,286	4,552	4,361	9,817	2,708
RQ1	`QUERY AND ('documenti*' OR 'documenta*')`	138	32	29	143	18
RQ2	`QUERY AND (tool*)`	465	68	89	485	29

Table 5.6: Automated search results per RQ and abstract, keywords or title

RQ	Found	Studies
RQ1	38	[S118], [S119], [S120], [S121], [S74], [S122], [S123], [S124], [S112], [S125], [S126], [S116], [S127], [S128], [S129], [S130], [S131], [132], [S133], [S75], [S134], [S135], [S115], [S136], [S117], [S137], [S138], [S139], [S76], [S111], [S140], [S141], [S142], [S143], [S144], [102], [S31], [S145]
RQ2	18	[S146], [147], [S148], [S118], [S121], [S149], [S150], [S86], [S151], [S152], [S153], [S134], [S136], [S114], [S140], [S154], [S155], [S97]

Table 5.7: Studies contributing to answering the research questions from the manual search

RQ	Found	Studies
RQ1	40	[S118], [S119], [S120], [S121], [S74], [S122], [S123], [S124], [S112], [S125], [S126], [S116], [S127], [S128], [S129], [S130], [S131], [132], [S133], [S75], [S134], [S135], [S115], [S136], [S117], [S137], [S138], [S139], [S76], [S31], [S111], [S140], [S141], [S156], [S142], [S143], [S144], [102], [S31], [S145]
RQ2	23	[S146], [147], [S148], [S118], [S157], [S121], [S149], [S150], [S86], [S151], [S158], [S152], [S153], [S134], [S159], [S136], [S160], [S114], [S140], [S154], [S155], [S161], [S97]

Table 5.8: Studies contributing to answering the research questions from the database search

5.3.2 Classification Scheme Using a Systematic Map

As a next step in the analysis, we classified the studies using a systematic map, as described by Petersen, Vakkalanka, and Kuzniarz [94] and Petersen, Feldt, Mujtaba, *et al.* [103]. Each study was categorized using three facets: 1) a contribution facet covering the type of contribution to the software engineering domain, 2) a research type facet describing the type of study and 3) a context facet that maps the content of the studies to our research questions. Figure 5.7 shows the resulting map using two bubble charts. The size of the bubble represents the number of studies falling into the corresponding categories. The absolute number of studies is shown in the centers of the bubbles followed by a letter that refers to the list of studies that can be found

RQ	Found	Studies
RQ1	40	[S118], [S119], [S120], [S121], [S74], [S122], [S123], [S124], [S112], [S125], [S126], [S116], [S127], [S128], [S129], [S130], [S131], [132], [S133], [S75], [S134], [S135], [S115], [S136], [S117], [S137], [S138], [S139], [S76], [S31], [S111], [S140], [S141], [S156], [S142], [S143], [S144], [102], [S31], [S145]
RQ2	23	[S146], [147], [S148], [S118], [S157], [S121], [S149], [S150], [S86], [S151], [S158], [S152], [S153], [S134], [S159], [S136], [S160], [S114], [S140], [S154], [S155], [S161], [S97]

Table 5.9: Final set of studies that contribute to answering the research questions

Year	RQ1	RQ2	Total
2001	1	0	1
2002	1	0	1
2003	3	0	3
2004	0	1	1
2005	3	0	3
2006	0	0	0
2007	0	2	2
2008	4	5	9
2009	0	4	4
2010	5	1	6
2011	4	2	6
2012	4	2	6
2013	5	2	7
2014	9	3	12
2015	1	2	3
2016	6	8	14
2017	12	5	17
2018	2	7	9
Total	**60**	**44**	**104**

Table 5.10: Publications per year for RQ1, RQ2 and totals per year and RQ

in Appendix 10. For example, in the coordinate plane between the *Context Facet* "Documentation subjects: architecture" and *Research Facet* "Solution Proposal", the bubble represented by the letter "k" shows that 17 studies have been found. Studies can appear in multiple facets. The total number of 58 unique studies has been mapped 200 times.

For the contribution and research facets, we used existing classification schemes by Petersen, Feldt, Mujtaba, *et al.* [103], and Wieringa, Maiden, Mead, *et al.* [162], respectively. The classification scheme for the **Contribution facets** from Peterson [103] describes the potential categories of a paper's contribution: Metric, Tool, Model, Method, and Process. For our Systematic Mapping Study, we moved the Tool category to the Context facet, because RQ2 concerns tools. The classification scheme for the **Research facets** from Wieringa, Maiden, Mead, *et al.* [162] includes six types, four of which were found in our primary studies:

1. **Validation Research.** The investigation of a problem or implementation of a technique in practice.
2. **Evaluation Research.** The validation of a solution proposal that has not yet been investigated.
3. **Solution Proposal.** This type of papers contains solution proposals without validation.
4. **Experience Papers.** This type of papers describe *what* has been experienced by the author as matters of fact, and do not describe the reasons *why*.

The categories of the context facet evolved while doing the data extraction. We merged categories where appropriate to keep the number of categories small so we could plot them against the other two facets. In the following, those categories are briefly described. Additionally, we assign each category to one of the research questions:

1. *Documentation life cycle*: aspects of creation, maintenance and management of documentation artifacts (RQ1).
2. *Documentation subjects: architecture*: architecture related documentation such as design, solutions and architecture description (RQ1).
3. *Documentation subjects: source-code*: the documentation of source-code and source-code related aspects

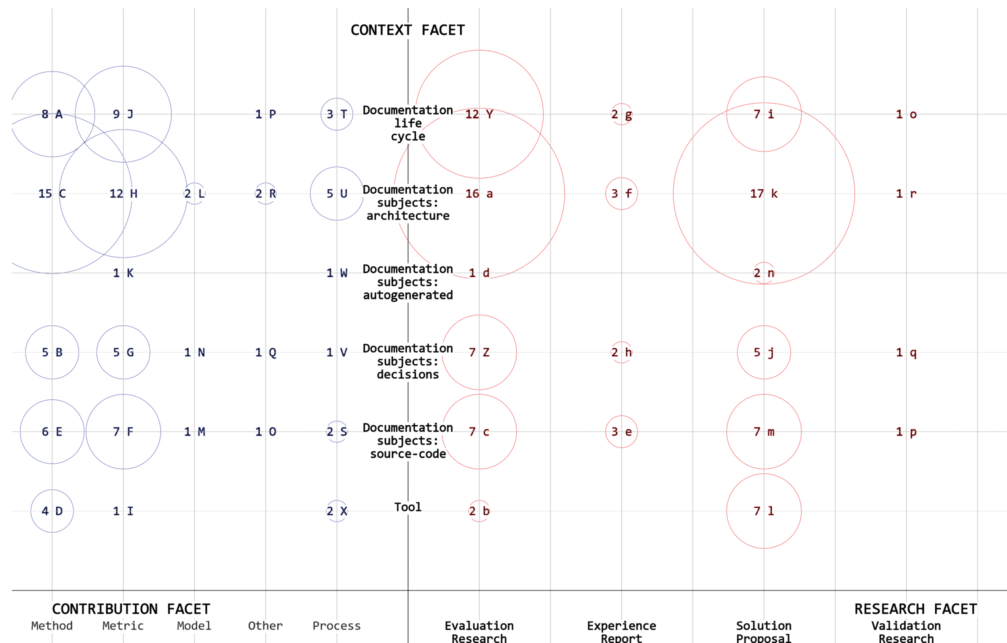


Figure 5.7: Mapping of classification facets with papers

such as version control (RQ1).

4. *Documentation subjects: auto generated documentation*: the storing and retrieval of documentation that is scattered throughout a software ecosystem and is stored in tools such as git commits, Jira tasks or wiki-like documents (RQ1). Please note that we omitted the term "documentation" in the bubble chart to ease readability.
5. *Documentation subjects: decisions*: software architecture decisions and their rationale (RQ1).
6. *Tool*: how tools are used in supporting documentation in continuous software development (RQ2).

Figure 5.7 shows that architecture documentation is a popular topic within the studies (42 papers in total), predominantly as solution proposals (21 papers) and evaluation research (18 papers). The documentation life cycle is also found in many studies (28 papers), as well as source-code documentation (25 papers), again primarily in the shape of solution proposals or evaluation research.

The most frequent contribution types of the identified studies are method (44 papers), metrics (35 papers), and models (27 papers); all three are mostly found on architecture documentation. It is notable that the least number of studies map to the tool category, which we consider counter-intuitive as continuous software development is a discipline that makes vast use of tool-ecosystems and automation.

5.3.3 Results for RQ1: Documentation Challenges and Practices

This section describes the results of our analysis on studies assigned to RQ1 (*What are documentation practices and resulting challenges in CSD?*). Table 5.9 lists all studies considered in this analysis. Continuous software development, as mentioned in the Introduction section, is not a development process model on its own; it is rather an umbrella term for existing methods that share certain characteristics. The papers found in this mapping study cover the following specific process models and methods:

- Lean Software Development [2],
- Scrum [1],
- Extreme Programming (XP)[S163],
- Feature-Driven Development (FDD) [S164],
- Crystal Clear [S165],
- Adaptive Software Development (ASD) [S166],
- Dynamic systems development method (DSDM) [102], [S167],
- Microsoft Solution Framework (MSF) [102], [S168],
- Agile Unified Process (AUP) [102], [169],
- and Test Driven Development [S60].

In the context of these process models and methods, we identified the **documentation challenges** listed below. By documentation challenges, we refer to obstacles that developers face towards documenting knowledge

about the system or keeping it up to date.

1. **Informal documentation is hard to understand.** As stated above, the sparse written documentation in CSD is often informal and volatile (e.g. white board sketches and drawings [S138], [S126]). Prashant Gandhi, Haugen, Hill, *et al.* [S126] refer to the different backgrounds from architects, reviewers and other stakeholders and note that it is rather cumbersome for one stakeholder to understand and improve the informal documentation of another [S126]. Such types of informal documentation artifacts require a kind of *"voice-over"* or additional explanation to be effective for knowledge transfer[S139].
2. **Documentation is considered waste.** Generally, documentation is considered waste when it does not contribute to the end product [2], [S135]. Documentation is only created if it is required to create the end product, or to raise the quality of the end product. Prause and Durdik [S135] differentiate between documentation for developers and for end-users. Documentation for developers does not contribute to the end product and is therefore neglected. The source code itself is considered the "ultimate documentation". An example of documentation that does contribute to the end product is a user-manual, for instance [S135]. As a result, design knowledge, reasoning knowledge, as well as knowledge about the problem space are typically not preserved in CSD in any written form.
3. **Productivity is measured by the amount of working software only.** In CSD, productivity is measured by the amount of delivered working software over development time. Beck and other founders of the agile manifesto state that working software is valued over comprehensive documentation [1], [S145], [170], [S171]. Thus, they emphasize that working code is the ultimate measure of productivity; documentation has value, but its comprehensiveness is less important. Stettina and Heijstek [S138] note that documentation is rather seen as a burden, than a (co-)created artifact [S138]. This attitude causes developers to generally consider documentation as counter-productive, which in turns causes knowledge loss.
4. **Documentation is out-of-sync with the software.** In CSD, developers don't keep documentation in sync with the actual software [S172]. This applies to both documentation outside the code such as in Microsoft Word documents and wiki-like tools, but also to source code documentation, e.g. regarding the objectives of methods or their parameters. Especially source code documentation is often outdated because CSD emphasizes the continuous update of code, but not its documentation. This is an issue, as stakeholders lose confidence and trust in the documentation [S126], which makes the sparse documentation even less useful. A lack of up-to-date documentation is particularly problematic in the context of architecture design decisions, as it leads to a loss of rationale behind design choices and considered alternatives; it thus become increasingly difficult to understand and judge solutions during software evolution [S173].
5. **Short-term focus.** Producing comprehensive documentation is a resource-intensive task that interferes with other short-term tasks like sketching diagrams or programming. Primarily, the short term goals of design, programming or maintenance tasks can be achieved without documenting important decisions, documenting rationale, consequences or alternatives [174], [S76]. However, this focus on achieving mostly short-term goals has an adverse effect: all the knowledge that is required for those goals disappears over the following iterations with changing context, new objectives and new team members. Nawrocki, Jasinski, Walter, *et al.* [S75] state that in XP there are three sources of knowledge about the software that are required but are hard to maintain in the long run [S75]: the code, test cases and the memory of the developers.

Despite, the aforementioned challenges, we were also able to observe **documentation practices**. In this study, a "practice" is defined as an activity that is usually or regularly conducted, e.g. as a habit, tradition, rule, or organizational culture.

1. **Non-written and informal communication.** In CSD, *verbal communication* is often used to achieve a mutual understanding between team members (e.g. in [S163]) rather than written documentation. Verbal communication is also one of the twelve principles in the agile manifesto [1], which states that face-to-face communication is both most effective and most efficient within a development team. For agile development in general and XP in particular, Prause and Durdik [S135] state that *knowledge is the result of collaboration* and is spread by different means ([S135]), other than written documentation. Often only *sketches and informal drawings* are used to support the verbal communication. One exception to this rule is requirements, which are typically documented in the format of user stories [175]. The sparse documentation that is deliberately created for documentation purposes is usually *created afterwards* and describes the state of the software "as is", rather than the software "to be" [S143]; this has the advantage of being up-to-date.
2. **Usage of development artifacts for documentation purposes.** Apart from artifacts created solely for the purpose of documentation, some artifacts created as part of the development process can also

serve as a type of documentation. TDD and BDD [S31], for instance, lead to executable specifications of the software to be built [S176]. Another form of executable documentation is "infrastructure as code", as mentioned by Callanan and Spillane [S177]. Infrastructure-as-code refers to any executable description of the infrastructure that is not part of the application itself [73]. This can be achieved with tools like Ansible, or Puppet, for instance.

3. **Architecture frameworks** Although architecture knowledge often evaporates in CSD projects, we have seen one particular documentation format, namely architecture frameworks, being used in practice. We briefly describe two examples of frameworks that qualify as architecture frameworks according to the definition in ISO/IEC/IEEE 42010 [90], while specifically addressing concerns of continuous software development. Di Nitto, Jamshidi, Guerriero, *et al.* [S125] proposed a framework called SQUID (Specification Quality In DevOps), an extension of Kruchten's 4+1 view model [S178] with additional viewpoints for dealing with DevOps-related concerns [S125]: an Operations Viewpoint, a Monitoring Viewpoint, a Deployment Viewpoint, and a Quality Verification Viewpoint. Similarly, the continuous integration and delivery architecture framework (Cinders), described byStåhl and Bosch [S179], is an architecture framework specifically designed to deal with architectural concerns in continuous integration and delivery [S179].

5.3.1.1 Interpretations of the Results

We think that challenges lead to practices, and vice versa, as illustrated in Figure 5.8. The first relation

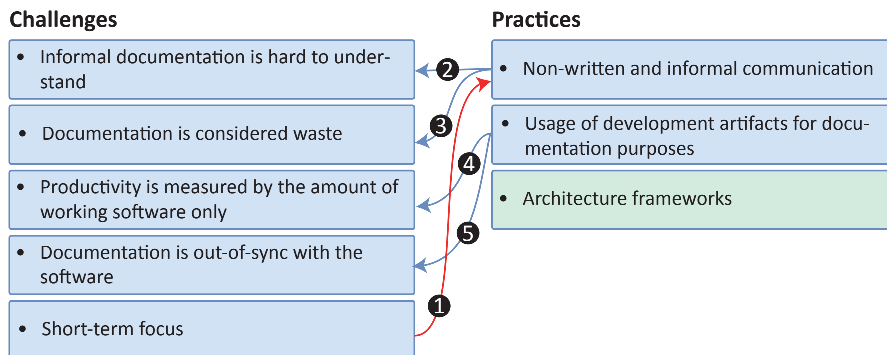


Figure 5.8: Possible relations between challenges and practices. The green box indicates a positive effect on the contribution to better documentation.

is between the challenge of short-term focus (1) that leads to non-written and informal communication. In turn, the non-written and informal communication leads to documentation that is hard to understand (2), and documentation being considered waste when it does not contribute to the end product (3). Using development artifacts for documentation purposes leads to the challenges that productivity is measured by the amount of software only (4) or that documentation is out-of-sync with the software (5). Furthermore, only the practice of architecture frameworks might be considered a practice to contribute to better documentation (see the green box in Figure 5.8).

5.3.1.2 Implications for Practitioners

The demand for fast time-to-market leads to fewer artifacts with lower quality. In small teams that are geographically located in one building or one room with long-term employees, informal knowledge is built up in the team. For larger teams, geographically distributed or with changing team members, building up knowledge about the software product and processes might be challenging. A typical practice for documentation is that a whiteboard sketch, and formal API documentation are considered sufficient instead of big upfront documentation with many UML diagrams. Apparently, these informal documentation practices are just enough to start an iteration. At the same time, however, these practices are not sufficient for operations, maintenance, or knowledge transfer. Another candidate approach to overcome these challenges is executable documentation; for TDD, this is a common practice.

5.3.1.3 Areas for Future Research

Future research is required to investigate if just enough upfront documentation can be limited to shaping thoughts by using informal whiteboard sketches, together with the codified API documentation. The upfront documentation should be accompanied by design-when-done after an iteration is completed. Anything that can be generated or reverse-engineered is not required to document because it is available anytime. Relevant for operations, maintenance, and knowledge transfer are decisions, considerations about the software product and process, and team organization.

A second area for future research is executable documentation. The practice of TDD, which is one example of executable specification, is a non-intrusive way of documenting requirements as part of the development process. This, however may not be the case for other types of executable specifications. In general, the question how executable documentation can be best produced and consumed by developers is subject to further research.

5.3.4 Results for RQ2: Tools Used in CSD

In RQ2, we investigated, *which tools are used in CSD with respect to documentation.* The studies we considered for answering this research question are listed in Table 5.9. The studies from the final result are presented in Figure 5.7 and Appendix 10. As already discussed, the sparse documentation in CSD is scattered over the entire tool ecosystem, mainly in the form of executable specifications. This concerns up-front documentation (mainly requirements), as well as design, code, and deployment information. Table 5.11 lists tools typically used in CSD practices for documentation purposes, along with the *type* of documentation (mark-down, binary, drawings), and *what* is documented (decisions, annotations, commit messages).

An ideal tool for documentation, according to Cannizzo, Marcionetti, and Moser [S180], would support four main practices: status visibility (build tools), extensive automation (i.e. automating as much as possible), effective communication (collaboration and communication tools, metrics tools), and tools for immediate feedback (test tools) [S180]. Other researchers have proposed simple solutions that can be readily used. For preserving reasoning information, Borrego, Morán, Palacio Cinco, *et al.* [S74] suggest a simple tagging mechanism [S74]. Buchmann [S181] propose a tool for automatically transforming handwritten sketches on paper and whiteboards to high-fidelity UML drawings (tool: Valkyrie) [S181]. Aguiar [S146] prefers wikis for documentation, because wikis and agility share goals like simplicity, flexibility, and open collaboration, thus being natural documentation tools to agile projects [S146]. Waits and Yankel [S140] observe that binary files, such as Microsoft Word or PowerPoint files, are hard to maintain, hamper velocity and can't make efficient use of a version control system (VCS), because changes cannot efficiently be tracked [S140].

Tool	Short description	Type of information	What is documented?	Studies
Word, Excel, PowerPoint etc.	Used for reports and intended for long term usage	Binary files	Decisions, drawing, sketches, pictures of whiteboard drawings	[S180],[S137], [S182], [S120], [S121], [S74], [S183], [S126], [S184], [S134], [S135], [S159], [S138], [S76], [S32], [S114], [S185], [S140], [S186], [S187], [S188], [S180], [189], [S141], [S190], [S191], [S142], [S143], [S192], [S193], [S144], [S167], [S194], [S195], [S196], [S197], [S198], [S199], [S31], [S200]
Wiki	Wiki-likes, Confluence	Short descriptions in the format of the tool, e.g. xml, html	Tasks, how-to's, SRS, SAD, SDD, info on PoCs, Prototypes, Releases	[S120], [S121], [S74], [S183], [S126], [S184], [S134], [S135], [S159], [S76], [S32], [S114], [S185], [S140], [S186], [S187], [S180], [189], [S141], [S190], [S191], [S192], [S193], [S144], [S194], [S196], [S197], [S198], [S199], [S31], [S200], [S137], [S138], [S182], [S188], [S142], [S143], [S167], [S195]
Source control	Tools for Git, Mercurial, SVN	Source-code, annotations, commit messages	Source-code, commit messages	[S138], [S140], [S142], [S136], [S74],
Scripts	Executable lines of code to run tasks	Human readable text	Infrastructure-as-code: Tests, integration, and deployment including installation, configuration, data import, and security	[201], [S124], [S76], [S141], [S142], [S32], [S31], [S131], [S145]

Continued on next page

Table 5.11 – continued from previous page

Tool	Short description	Type of information	What is documented?	Studies
Markdown editors	Light weight text editor, often used without editor but edited directly	Human readable text	Anything that can be documented in Word, Excel, PowerPoint, or Wikis including configuration files	[S140], [S202]
Verbal communication	The proverbial water-cooler conversation.	Face-to-face	No document, knowledge remains tacit	[S143], [S138], [S142], [S31], [S128], [102], [S145]

Table 5.11: Documentation Tools in CSD retrieved from the studies for RQ2.

Apart from tools used for the purpose of documentation, many tools used in CSD for other purposes also have documentary value. Kersten found that the number of different tools used in CSD is rapidly growing. He explains this phenomenon with a "democratization" of the toolchain, i.e. practitioners choose their own tools for different tasks rather than being obliged by a top-down control model for the tool ecosystem [S97]. This is also the case for documentation. There is no one-size-fits-all documentation tool; on the contrary, practitioners in CSD document what they like, wherever the like.

In the following, we discuss such CSD tools that can be used for documentation purposes. Specifically, we present a list of tool categories together with the type of documentation information associated with each category. The list is compiled from documentation usages found in four primary studies: Kersten presents a landscape for tools and tool-categories [S97]; Partial tool-chains are presented by Poth, Werner, and Lei [203], and Wettinger, Breitenbücher, Kopp, *et al.* [S155], who both focus on tools used in CI/CD pipelines; Mäkinen, Leppänen, Kilamo, *et al.* [204], present elements of a modern development toolchain [204].

1. **Development tools**

   (a) **Requirements management tools** (e.g. Blueprint, RequirementONE)
   The documentation information includes stakeholder concerns, risks, constraints and context formulated as *specifications*: these are typically codified instructions, sufficient for developers to start an iteration. Such specifications range from very informal and abstract (e.g. user stories and high-level use-case descriptions) to formal and concrete (e.g. detailed use-case descriptions with pre- and post-conditions, or Cucumber in combination with Gerkin).

   (b) **IDEs** (e.g. IntelliJ, Eclipse, Cloud9)
   The documentation information include the *source code* of the software, often *annotated* with comments, meant for developers to understand the code. The annotations in the code are also used for the automated generation of documentation for APIs.

   (c) **Agile Management tools** (e.g. Active.collab, Agile bench, JIRA)
   Agile management tools are used to support developers in applying agile methods like Scrum, Lean, or Kanban. The tools typically capture information about requirements (e.g. user stories or use-cases), tasks, progress (e.f. burn-down charts), planned and achieved goals for iterations, development speed (e.g. team velocity), and the provide traceability between requirements, tasks, and code (e.g. a JIRA board on which tasks fall under use-cases and are linked to source-code using Git branches and pull requests).

   (d) **Development Analytics tools** (e.g. SonarQube, Metrixware)
   The documentation information includes metrics, as well as *actionable data* that can be used by all roles in the software development including managers, developers, and maintainers. Typical examples of actionable data are quality measurements in SonarQube (a static source code analysis tool) along with concrete suggestions for fixing the quality rule violations.

   (e) **Repositories / Development Communities** (e.g. GitLab, StackShare, StackOverflow)
   The documentation information that repositories hold are the *source code* ready for review, or deployment and *commit messages* where every push to the repository (ideally) includes a meaningful comment on the changes. In development communities (e.g. Stack Overflow of GitHub), the information contains questions and answers for software development topics. As such, it can be considered as a *knowledge base*.

2. **Test tools**

   (a) **Performance / Load / Stress test tools** (e.g. JMeter and SmartStorm)
   These types of test codify specific QoS requirements (typically quality attribute requirements in the categories of performance efficiency, reliability, and security (see ISO/IEC/IEEE 25010 ([205]).

   (b) **Functional Automation, Virtualization tools** (e.g. Cucumber and Appium)
   The documentation information includes: the *test object* (method, component, API, UI); the test *strategy* with guiding values, principles, and objectives for stakeholders; and test *tactics, and types*

with techniques (e.g. destructive/UI tests), processes (e.g. CI/CD) and approaches (e.g. manual/automated testing) applied to specific test *tasks*.

(c) **Continuous testing tools** (e.g. test.ai, buildbot)
Continuous testing is a software testing type that involves a process of testing early, often, everywhere, and automatically (to the best possible extent). The objective of continuous testing is to find defects and support immediate response to the defects during the whole development cycle, including requirement specification, development, and maintenance. The documentation information refers to any phase that delivers *measurable software* or *software artifacts*. Test-specifications written for automation purposes (e.g. unit-tests, integration tests and automated end-to-end tests) are functional specifications for source-code units. They can be seen as formal specifications of functional requirements. Often, test-cases also cover QoS-parameters, e.g. the maximum accepted response time of a rest-endpoint.

(d) **Service Virtualization testing tools** (e.g. Smartbear, Parasoft)
Service virtualization is used when the system makes use of an API that is not controlled by the development team. The Service virtualization emulates behavior of the external system, external APIs, cloud-based applications, service-oriented architectures or micro-services that are out of control of the development team. This documentation information includes data, (non-)functional tests or behavior that *emulates the external system*.

3. **Deployment tools**

(a) **App Automation tools** (e.g. Ansible, Puppet, Chef)
The documentation information includes instructions for installation, updates and configuration of software, the import of data, and hardening of systems. This information is typically defined in CI/CD scripts. This type of scripts assists in the automation of (complex) IT tasks into *repeatable playbooks*. Although the scripts contain information for a range of tasks, they are typically configured for a single task such as installation or phase such as test.

(b) **CI/CD** (e.g. CircleCI, Jenkins)
The documentation information includes the *relation between single tasks* and the *results of executing these tasks*. The automation considers the execution of single tasks from App Automation into a set of scripts for multiple tasks (e.g. installation, configuration, import, hardening) and for multiple stages (e.g. development, test, integration, deployment) whether on premise or in the cloud. The test results show developers or release managers the sanity of the builds in a comprehensive visual overview.

4. **Service execution tools**

(a) **Cloud / Container Orchestration / Management** (e.g. Docker, Mesos)
The documentation information includes *metrics* about non-functional requirements such as, but not limited to availability and reliability. It includes also installation and configuration scripts for the software as well as scripts for automated up- or down scaling. This can refer to a single container as well as the orchestration of containers. Infrastructure monitoring tools provide visibility of the complete infrastructure and allow for troubleshooting and resource optimization.

5. **Monitoring tools**

(a) **Monitoring & Management** (e.g. DataDog, RunDeck)
In CSD, a lot of processes, and (supporting) applications run at the same time which can lead to a chaotic software development ecosystem as well as a hard to manage deployment pipeline and production environment. The monitoring and management tools support the team to have control of processes and software products. The documentation information captured in the tools contains desired and actual quality-of-service parameters, as well as standard operating procedures for incidents.

6. **Security tools**

(a) **Container Security** (e.g. AppArmor, Cloud Insights)
The documentation information for container security involves the *build scripts* for the container and the additional *security policies* (such as non-root user, application isolation, and authentication/authorization).

(b) **Application Security** (e.g. Threat Stack, HyTrust)
Containerized applications typically make use of a micro-service architecture. The information comprises infrastructural security, information on the security aspects (confidentiality, integrity, non-repudiation, accountability and authenticity), information on the distribution of the multiple applications in multiple containers including functionality, data, subsystems, and APIs.

(c) **DevSecOps** (e.g. Cigital, CheckMarx)

DevSecOps refers to the processes and practices to merge the security that is used in development process into processes in operations and vice versa with the purpose of faster deployment with security measures in place. The documentation information includes authentication, and authorization with *roles for users* (where applications are also defined as a user that needs to be authenticated to obtain authorization). It also includes information about *technical (software and hardware) security*, as well as *test procedures* to force and validate security measures.

7. **API management tools and directories**
   (a) **API management tools** (e.g. Smartbear, Mashape, RapidAPI, OpenAPI)
   These tools document the definition of the *resource description, endpoints with methods, parameters*, often a *request and response example*, and sometimes a *playground* for testing the API.
   (b) **API directories** (e.g. ProgrammableWeb)
   These provide a *directory of external APIs* that include a description, documentation for developers, SDK, "How to" instructions, (optional) libraries, and information from the community and thus also capture general documentation information about APIs and underlying technologies.

Apart from tools used for the purpose of documentation, many tools used in CSD for other purposes also have documentary value. Kersten [S97] found that the number of different tools used in CSD is rapidly growing. He explains this phenomenon with a "democratization" of the tool-chain, i.e. practitioners choose their own tools for different tasks rather than being obliged by a top-down control model for the tool ecosystem [S97]. This is also the case for documentation. There is no one-size-fits-all documentation tool; on the contrary, practitioners in CSD document what they like, wherever the like.

In the following, we discuss such CSD tools that can be used for documentation purposes. Specifically, we present a list of tool categories together with the type of documentation information associated with each category. The list is compiled from documentation usages found in four primary studies: Kersten [S97] presents a landscape for tools and tool-categories [S97]; Partial tool-chains are presented by Poth, Werner, and Lei [203], and [S155] , who both focus on tools used in CI/CD pipelines; Mäkinen, Leppänen, Kilamo, *et al.* [204], present elements of a modern development tool-chain [204].

1. **Development tools**

   (a) **Requirements management tools** (e.g. Blueprint, RequirementONE)
   The documentation information includes stakeholder concerns, risks, constraints and context formulated as *specifications*: these are typically codified instructions, sufficient for developers to start an iteration. Such specifications range from very informal and abstract (e.g. user stories and high-level use-case descriptions) to formal and concrete (e.g. detailed use-case descriptions with pre- and post-conditions, or Cucumber in combination with Gerkin).
   (b) **IDEs** (e.g. IntelliJ, Eclipse, Cloud9)
   The documentation information include the *source code* of the software, often *annotated* with comments, meant for developers to understand the code. The annotations in the code are also used for the automated generation of documentation for APIs.
   (c) **Agile Management tools** (e.g. Active.collab, Agile bench, JIRA)
   Agile management tools are used to support developers in applying agile methods like Scrum, Lean, or Kanban. The tools typically capture information about requirements (e.g. user stories or use-cases), tasks, progress (e.f. burn-down charts), planned and achieved goals for iterations, development speed (e.g. team velocity), and the provide traceability between requirements, tasks, and code (e.g. a JIRA board on which tasks fall under use-cases and are linked to source-code using Git branches and pull requests).
   (d) **Development Analytics tools** (e.g. SonarQube, Metrixware)
   The documentation information includes metrics, as well as *actionable data* that can be used by all roles in the software development including managers, developers, and maintainers. Typical examples of actionable data are quality measurements in SonarQube (a static source code analysis tool) along with concrete suggestions for fixing the quality rule violations.
   (e) **Repositories / Development Communities** (e.g. GitLab, StackShare, StackOverflow)
   The documentation information that repositories hold are the *source code* ready for review, or deployment and *commit messages* where every push to the repository (ideally) includes a meaningful comment on the changes. In development communities (e.g. Stack Overflow of GitHub), the information contains questions and answers for software development topics. As such, it can be considered as a *knowledge base*.

2. **Test tools**
   (a) **Performance / Load / Stress test tools** (e.g. JMeter and SmartStorm)

These types of test codify specific QoS requirements (typically quality attribute requirements in the categories of performance efficiency, reliability, and security (see ISO/IEC/IEEE 25010 ([205]).

(b) **Functional Automation, Virtualization tools** (e.g. Cucumber and Appium)
The documentation information includes: the *test object* (method, component, API, UI); the test *strategy* with guiding values, principles, and objectives for stakeholders; and test *tactics, and types* with techniques (e.g. destructive/UI tests), processes (e.g. CI/CD) and approaches (e.g. manual/automated testing) applied to specific test *tasks*.

(c) **Continuous testing tools** (e.g. test.ai, buildbot)
Continuous testing is a software testing type that involves a process of testing early, often, everywhere, and automatically (to the best possible extent). The objective of continuous testing is to find defects and support immediate response to the defects during the whole development cycle, including requirement specification, development, and maintenance. The documentation information refers to any phase that delivers *measurable software* or *software artifacts*. Test-specifications written for automation purposes (e.g. unit-tests, integration tests and automated end-to-end tests) are functional specifications for source-code units. They can be seen as formal specifications of functional requirements. Often, test-cases also cover QoS-parameters, e.g. the maximum accepted response time of a rest-endpoint.

(d) **Service Virtualization testing tools** (e.g. Smartbear, Parasoft)
Service virtualization is used when the system makes use of an API that is not controlled by the development team. The Service virtualization emulates behavior of the external system, external APIs, cloud-based applications, service-oriented architectures or micro-services that are out of control of the development team. This documentation information includes data, (non-)functional tests or behavior that *emulates the external system*.

3. **Deployment tools**

(a) **App Automation tools** (e.g. Ansible, Puppet, Chef)
The documentation information includes instructions for installation, updates and configuration of software, the import of data, and hardening of systems. This information is typically defined in CI/CD scripts. This type of scripts assists in the automation of (complex) IT tasks into *repeatable playbooks*. Although the scripts contain information for a range of tasks, they are typically configured for a single task such as installation or phase such as test.

(b) **CI/CD** (e.g. CircleCI, Jenkins)
The documentation information includes the *relation between single tasks* and the *results of executing these tasks*. The automation considers the execution of single tasks from App Automation into a set of scripts for multiple tasks (e.g. installation, configuration, import, hardening) and for multiple stages (e.g. development, test, integration, deployment) whether on premise or in the cloud. The test results show developers or release managers the sanity of the builds in a comprehensive visual overview.

4. **Service execution tools**

(a) **Cloud / Container Orchestration / Management** (e.g. Docker, Mesos)
The documentation information includes *metrics* about non-functional requirements such as, but not limited to availability and reliability. It includes also installation and configuration scripts for the software as well as scripts for automated up- or down scaling. This can refer to a single container as well as the orchestration of containers. Infrastructure monitoring tools provide visibility of the complete infrastructure and allow for troubleshooting and resource optimization.

5. **Monitoring tools**

(a) **Monitoring & Management** (e.g. DataDog, RunDeck)
In CSD, a lot of processes, and (supporting) applications run at the same time which can lead to a chaotic software development ecosystem as well as a hard to manage deployment pipeline and production environment. The monitoring and management tools support the team to have control of processes and software products. The documentation information captured in the tools contains desired and actual quality-of-service parameters, as well as standard operating procedures for incidents.

6. **Security tools**

(a) **Container Security** (e.g. AppArmor, Cloud Insights)
The documentation information for container security involves the *build scripts* for the container and the additional *security policies* (such as non-root user, application isolation, and authentication/authorization).

(b) **Application Security** (e.g. Threat Stack, HyTrust)

Containerized applications typically make use of a micro-service architecture. The information comprises infrastructural security, information on the security aspects (confidentiality, integrity, non-repudiation, accountability and authenticity), information on the distribution of the multiple applications in multiple containers including functionality, data, subsystems, and APIs.

(c) **DevSecOps** (e.g. Cigital, CheckMarx)
DevSecOps refers to the processes and practices to merge the security that is used in development process into processes in operations and vice versa with the purpose of faster deployment with security measures in place. The documentation information includes authentication, and authorization with *roles for users* (where applications are also defined as a user that needs to be authenticated to obtain authorization). It also includes information about *technical (software and hardware) security*, as well as *test procedures* to force and validate security measures.

7. **API management tools and directories**

(a) **API management tools** (e.g. Smartbear, Mashape, RapidAPI, OpenAPI)
These tools document the definition of the *resource description, endpoints with methods, parameters*, often a *request and response example*, and sometimes a *playground* for testing the API.

(b) **API directories** (e.g. ProgrammableWeb)
These provide a *directory of external APIs* that include a description, documentation for developers, SDK, "How to" instructions, (optional) libraries, and information from the community and thus also capture general documentation information about APIs and underlying technologies.

As shown in the list above, documentation information is scattered throughout an entire eco-system of tools rather than being provided in a single self-contained document. As a consequence of the scattering, stakeholders who need to understand the vision and long term goals, architecture decisions, risks, constraints, interface definitions, deployment instructions etc., need to dig into the entire tool-stack [S112], [S126]. There is no single source of truth, but there are many sources of truth, each holding information on a relevant part of the software product ([98]).

5.3.4.1 Interpretations of the Results

There are many tools used in CSD, for every phase (e.g. design, implementation, and testing), as well as every activity (e.g. drawing, collaborating, writing, and constructing). The amount of structure of the information is strongly related to the tool. Some information is easy to capture and easy for human communication such as whiteboard sketches or conversations in chats. At the same time, these types of information are hard for automatic processing. Source code on the other hand, can be automatically processed.

5.3.4.2 Implications for Practitioners

For the construction of the software product, the tools support the developers. As such, the tooling has a positive effect on the productivity. However, with the increase of the number of tools, information about the software product, processes and organization is distributed across these tools.

5.3.4.3 Areas for Future Research

A candidate area for future research could be to organize the documentation into yellow pages (wiki, git, markdown) that contains references to relevant documentation for designated stakeholders. For instance PowerPoint slides for conveying ideas about the software product, design documentation for developers and infrastructure-as-code for operations.

5.4 Discussion

In this section, we interpret our results and provide implications for practitioners and researchers. To begin with the interpretation of the results, the software engineering discipline has always been struggling with documentation. Parnas, for instance, reported back in 1994, that documentation - if written at all- is usually poorly organized, incomplete and imprecise [206]. The human factors that caused this problem back then, are - to the same extent- responsible for the issues reported over documentation in Continuous Software Development today. The difference is that missing or poor documentation was traditionally seen as the result of negligence or even misconduct of developers; in continuous software development it is deliberately promoted to a desired behavior. In other words, CSD puts many obstacles in the way of properly documenting the different aspects of

created knowledge (e.g. considering documentation as waste, having a short-term focus, measuring productivity through working software only).

In CSD, with a few mentioned exceptions like documenting requirements, dedicated documentation (i.e. documentation that does not serve as development artifact also) is informal (e.g. white board sketches) and needs to be supported by face-to-face communication. This may not be ideal, but we argue that it can be an effective and efficient approach to support the design reasoning process. However, it cannot effectively preserve knowledge and thus not serve as documentation; this is not a surprise as informal artifacts are not created for the purpose of documenting, but for the purpose of supporting a design discussion.

Knowledge-preserving documentation that stands on its own requires a certain level of formalism and needs to be created for the purpose of describing something unambiguously. Such documentation is very rarely created in CSD projects. Thus, in our opinion, the documentation practices in CSD -or lack thereof- do not contribute to solving the traditional problems related to knowledge loss and missing information during maintenance activities. Unfortunately, we have not seen evidence of new or emerging practices that can alleviate this problem.

Although the results we found regarding dedicated documentation *practices* in CSD are sobering, there is also good news. With the rise of Lean, Agile and DevOps projects, we observe a drastic boost in *tool-ecosystems*, which mainly stems from the goal to automate software-related processes as much as possible. This also enables new ways of thinking about documentation. The specifications required for automating processes (we refer to them as executable specifications), at the same time serve as documentation of the process. This essentially urges us to refine Robert Martin's statement that the truth can only be found in the code: now the truth can also be found in test scripts, provisioning scripts, build pipeline configurations, and cloud platform configurations, to name just a few.

5.4.1 Characteristics of Executable Documentation

Executable specifications have a lot of potential for serving as documentation in CSD. Their characteristics are in line with the principles of CSD, and at the same time address the previously mentioned traditional problems that come with missing documentation. We highlight the following characteristics of executable documentation that require further research.

Types of Executable Documentation

Types of executable documentation include: requirements, such as "Specification by Example" [207], test-cases resulting from Domain-Driven-Design[208] or TDD [S115], [S142], [S176], [S209] , database scripts with Data Definition Language (DDL) and Data Manipulation Language (DML), deployment scripts with Ansible [S31] or infrastructure-as-code [210].

Executable Documentation is never Out-Of-Sync

Executable documentation is never out-of-sync, it's evolution is naturally connected to the evolution of the other parts of the software. Executable documentation does not just describe the software, but it is part of the software.

Executable Documentation can be Tested

Executable documentation can be tested. If it does not lead to the desired results, then something must be wrong. In that respect, executable documentation is just like source-code.

Executable Documentation is Non-Intrusive

The process of creating executable documentation is not intrusive, i.e. developers do not stop their work to take care of documentation; coding and documenting are part of the same task.

In future research, these items will be investigated. Questions remain, for example, how can software development teams use such executable specifications? This could include a considerable amount of unstructured (and unrelated) data.

5.4.2 Implications for Practitioners

In the following, we present some implications for practitioners who want to benefit from the potential of CSD to document the created knowledge.

Tools, Tool-stacks, and Software Development Ecosystems

Support your entire development process by a tool-chain that seamlessly supports all activities in the process. Eliminate manual or interactive steps in the development process to the greatest possible extent. Manuals for developers describing process steps to follow (or the need for such manuals) should be considered as bad smells [211] that should be transferred to executable specifications interpreted by tools. Executable specifications are always up-to-date and at the same time document processes in an unambiguous way that can be interpreted by both machines and humans.

Informal Sketches

Use informal sketches (that are minimally intrusive) to support your design reasoning process and discussions with team members. The reasoning process and discussions ultimately lead to decisions that are implemented (e.g. in source-code or executable specifications). Consider briefly documenting the rationale behind those decisions that may not be obvious to other stakeholders (including future developers). Examples of obvious decisions are choices of tools or combinations of tools that are very popular for certain purposes, e.g. the combination of Elasticsearch, Logstash, and Kibana for distributed logging and analytics.

Use of Version Control

Keep everything under version control. Use project management tools or wikis as a central entry point for all information related to the project; otherwise, stakeholders may easily get lost in the great amount of project locations, tools and URLs. Also consider providing high-level overviews of the designed sub-systems, and link the respective executable specifications to the sub-systems to facilitate access for stakeholders.

5.4.3 Future Research

In terms of research, the results of this mapping study have shown that documentation in CSD, has not yet gained the required attention by the research community[8]. In the following, we describe three areas for future research:

1. The individual tools in a CSD ecosystem are mostly created separately, thus having limited interoperability. However, the combination of information from different tools can be “more than the sum of its parts”, i.e. it can provide insights that capture a greater part of the system and life cycle. Thus research is required to establish traceability links between the different types of tools and intelligently combine information from different kinds of executable documentation in dashboards.
2. Traditional architecture documentation approaches seem to come in direct conflict with the identified documentation challenges in CSD. Research is required to develop architecture documentation and specification approaches that integrate seamlessly in CSD-practices. For example, architecture frameworks could be developed that tap the potential of executable specifications, while preserving design rationale, explaining architecture to stakeholders, linking design decisions to concerns and architectural requirements and providing an informational basis for architectural evaluation.
3. The high degree of automation offers rich sources of information that can be mined using *Mining Software Repository* techniques, or in general *Data Science*. Examples of questions that could be addressed using such approaches in CSD are:
   - What is the current technical debt in source code, testing, requirements or other types?
   - What design decisions are likely to be outdated soon?
   - What is the cost-benefit ratio of specific features?
   - What is the optimal point in time for refactoring a specific sub-system?

[8] In August 2019, " executable documentation" had the following results: ACM: 9; Google Scholar: 207; IEEE: 2; ScienceDirect: 15; SpringerLink: 35; Web of Science: 3

5.5 Threats to Validity

We use the framework of Ampatzoglou, Bibi, Avgeriou, *et al.* [212] that describes potentials threats to validity for secondary studies. Specifically this framework classifies threats to validity in three categories, as illustrated in Table 5.12.

Category	Description
Study Selection Validity	These threats can be identified in the initial search process where the set of candidate papers for primary studies is selected and the study filtering where the final set of primaries studies is determined. Typical examples are the selection of digital libraries, search string construction and study selection bias.
Data Validity	These threats can be identified in the data extraction phase (a data set is populated) and data analysis phase (the data set is qualitatively or quantitatively analyzed). Typical examples include data collection bias and publication bias.
Research Validity	These threats can be identified over the whole mapping study and concern the design of the research. Typical examples are generalizability, and coverage of research questions.

Table 5.12: Classification of validity threats (Ampatzoglou, Bibi, Avgeriou, *et al.* [212])

5.5.1 Study Selection Validity

Regarding the selection of digital libraries, we have to a large extent addressed this by including the most used digital libraries in this area (which are also commonly used in secondary studies in software engineering). The construction of the search string may lead to yielding too many primary studies or missing relevant studies. We mitigated this threat by calibrating the search string through the quasi-gold standard. Specifically the QGS was used to assess the performance of the search string and refine it until all primary studies of the QGS were returned from the search string. The QGS itself was built using the snowballing technique guidelines as proposed by Wolhin [107].

Furthermore, we have mitigated the risk of an arbitrary starting year, because it was related to the year of the publication of the Agile Manifesto. With this decision we excluded a historic overview of consecutive concepts that lead to the Agile Manifesto; however, we did not aim at such a historic overview but a systematic classification and thematic analysis of literature.

The threat for non-English papers was not mitigated; these papers were excluded. We did however address the threat of studies not being accessible: we made sure we could access all studies. The threat of duplicate articles was mitigated by filtering on the Document Object Identifier. If a study appeared in multiple digital libraries, then the publishers' digital library was used and the duplicate was ignored. We excluded gray literature and included only studies from peer reviewed journals, conferences or workshops to have more rigor. Finally, the potential bias of study inclusion/exclusion was mitigated by discussion among the authors and accordingly revising the inclusion/exclusion criteria.

5.5.2 Data Validity

The risk of retrieving a small sample was mitigated by constructing a search string that could zoom in from a domain with over approximately 35.000 studies to finally about 200 relevant papers to answer the research questions. The threat of choosing the correct variables to be extracted was addressed through extensive discussions between the authors. The threat of publication bias (the majority of identified primary studies coming from specific venues) was mitigated by using snowballing. Furthermore, we addressed the threat of inadequate validity of primary studies through the inclusion criteria by only looking at peer reviewed venues. The threat of biasing the classification schema is mitigated by going through several iterations to refine the RQs, and redefining the search string and the analysis process. The threat of researchers' bias was partially mitigated by doing the analysis with multiple researchers where research and review were different roles, and by using a combination of manual and automated search.

5.5.3 Research Validity

The threat of repeatability is mitigated by meticulously documenting the study protocol. In addition, the retrieved studies, search strings and data are all available on https://theotheunissen.nl/SMS. The threat of the chosen research method bias is mitigated by extensive discussions among the authors and the rationale of our decision is clearly described in the study design section. Furthermore, the authors have also discussed in multiple iterations the choice and coverage of the research questions. Regarding the generizability of our results, they are only applicable within the scope of documentation in continuous software development.

5.6 Conclusions

We conducted a systematic mapping study to investigate the documentation practices and challenges, as well as the tooling used in continuous software development (CSD). The study has provided an overview of the relevant primary studies and has listed a number of challenges, practices, and tools that pertain to documentation in CSD.

Section 5 elaborates on our findings regarding documentation challenges and practices (RQ1). The challenges include: informal documentation is hard to understand, documentation is considered waste when it does not contribute to the end product, productivity is limited to the measured amount of working software, documentation is easily out-of-sync with the actual code, and there is short term focus. The practices include: a significant amount of communications happens verbally and informally; there is a positive usage of development artifacts for documentation purposes, such as TDD; and the use of architecture frameworks might positively influence documentation quality.

Section 5 discusses an increasing number of tool categories and tools that can be used to support development, operations, and maintenance in CSD (RQ2).

CSD is a high-paced evolving and dynamic environment; without tools, development and deployment would not be possible. An interesting side-effect of the tooling that has not been adequately researched yet, is that with every tool that is being used, knowledge about the piece of software is stored, maintained and transferred as well. For example, commits in a repository describe the changes of the source code and test scripts in a test tool describe the required outcomes of software. Knowledge about the software is scattered throughout all the tools in a software ecosystem. There is not a single source of truth, but there are a lot of sources of truth, each holding a small piece of knowledge. The discovery of these pieces of knowledge has not been investigated and it could be interesting to do further research on how to locate and combine these information sources.

Finally, we identified several implications for practitioners regarding the use of executable specifications in combination with a high degree of automation. Additionally, we found that architecture frameworks streamlined for use in CSD and dashboards combining information from the entire development tool chain are important areas for future research.

Appendix A: Mappings in the Bubble Chart

Category	A: Method, Documentation life cycle
Papers	[S136], [S75], [S118], [S148], [S129], [S134], [S117], [S139]
Found	8

Category	B: Method, Documentation subjects: decisions
Papers	[S136], [S119], [S148], [S129], [S139]
Found	5

Category	C: Method, Documentation subjects: architecture
Papers	[S136], [102], [S118], [S119], [S153], [S123], [S120], [S31], [S148], [S128], [S134], [S124], [S76], [S117], [S139]
Found	15

Category	D: Method, Tool
Papers	[S136], [S119], [S120], [S128]
Found	4

Category	E: Method, Documentation subjects: source-code
Papers	[102], [S119], [S148], [S128], [S76], [S139]
Found	6

Category	F: Metric, Documentation subjects: source-code
Papers	[S112], [S140], [S126], [S114], [S133], [S150], [S137]
Found	7

Category	G: Metric, Documentation subjects: decisions
Papers	[S112], [S126], [S114], [S133], [S150]
Found	5

Category	H: Metric, Documentation subjects: architecture
Papers	[S112], [S140], [S126], [S114], [S133], [S151], [S111], [S74], [S150], [S134], [S137], [S97]
Found	12

Category	I: Metric, Tool
Papers	[S112]
Found	1

Category	J: Metric, Documentation life cycle
Papers	[S140], [S126], [S114], [S151], [S111], [S74], [S134], [S122], [S137]
Found	9

Category	K: Metric, Documentation subjects: autogenerated
Papers	[S150]
Found	1

Category	L: Model, Documentation subjects: architecture
Papers	[S135], [S133]
Found	2

Category	M: Model, Documentation subjects: source-code
Papers	[S133]
Found	1

Category	N: Model, Documentation subjects: decisions
Papers	[S133]
Found	1

Category	O: Other, Documentation subjects: source-code
Papers	[S31]
Found	1

Category	P: Other, Documentation life cycle
Papers	[S31]
Found	1

Category	Q: Other, Documentation subjects: decisions
Papers	[S31]
Found	1

Category	R: Other, Documentation subjects: architecture
Papers	[S31], [S116]
Found	2

Category	S: Process, Documentation subjects: source-code
Papers	[S138], [S114]
Found	2

Category	T: Process, Documentation life cycle
Papers	[S138], [S114], [S145]
Found	3

Category	U: Process, Documentation subjects: architecture
Papers	[S138], [S114], [S121], [S145], [S124]
Found	5

Category	V: Process, Documentation subjects: decisions
Papers	[S114]
Found	1

Category	W: Process, Documentation subjects: autogenerated
Papers	[S121]
Found	1

Category	X: Process, Tool
Papers	[S121], [S145]
Found	2

Category	Y: Evaluation Research, Documentation life cycle
Papers	[S136], [S140], [S126], [S75], [S114], [S151], [S111], [S74], [S129], [S134], [S122], [S137]
Found	12

Category	Z: Evaluation Research, Documentation subjects: decisions
Papers	[S136], [S112], [S126], [S114], [S133], [S150], [S129]
Found	7

Category	a: Evaluation Research, Documentation subjects: architecture
Papers	[S136], [S112], [S140], [S126], [S114], [S153], [S133], [S123], [S151], [S111], [S74], [S150], [S134], [S116], [S137], [S97]
Found	16

Category	b: Evaluation Research, Tool
Papers	[S136], [S112]
Found	2

Category	c: Evaluation Research, Documentation subjects: source-code
Papers	[S112], [S140], [S126], [S114], [S133], [S150], [S137]
Found	7

Category	d: Evaluation Research, Documentation subjects: autogenerated
Papers	[S150]
Found	1

Category	e: Experience Report, Documentation subjects: source-code
Papers	[102], [S31], [S148]
Found	3

Category	f: Experience Report, Documentation subjects: architecture
Papers	[102], [S31], [S148]
Found	3

Category	g: Experience Report, Documentation life cycle
Papers	[S31], [S148]
Found	2

Category	h: Experience Report, Documentation subjects: decisions
Papers	[S31], [S148]
Found	2

Category	i: Solution Proposal, Documentation life cycle
Papers	[S136], [S138], [S118], [S145], [S131], [S117], [S139]
Found	7

Category	j: Solution Proposal, Documentation subjects: decisions
Papers	[S136], [S112], [S119], [S133], [S139]
Found	5

Category	k: Solution Proposal, Documentation subjects: architecture
Papers	[S136], [S112], [S138], [S118], [S135], [S119], [S121], [S133], [S123], [S120], [S31], [S128], [S145], [S124], [S76], [S117], [S139]
Found	17

Category	l: Solution Proposal, Tool
Papers	[S136], [S112], [S119], [S121], [S120], [S128], [S145]
Found	7

Category	m: Solution Proposal, Documentation subjects: source-code
Papers	[S112], [S138], [S119], [S133], [S128], [S76], [S139]
Found	7

Category	n: Solution Proposal, Documentation subjects: autogenerated
Papers	[S121], [S131]
Found	2

Category	o: Validation Research, Documentation life cycle
Papers	[S126]
Found	1

Category	p: Validation Research, Documentation subjects: source-code
Papers	[S126]
Found	1

Category	q: Validation Research, Documentation subjects: decisions
Papers	[S126]
Found	1

Category	r: Validation Research, Documentation subjects: architecture
Papers	[S126]
Found	1

Appendix B: Input From Experts

The email we send to the experts had this content:

```
Dear reader,

I am conducting a systematic mapping study to research the
literature on documentation, tooling and existing
frameworks in continuous software development
(or: agile, lean, DevOps, CI/CD).

Your input as an academic researcher or industry
practitioner in this area is appreciated.

BACKGROUND
Documentation of software architecture, design, development and
operations have a long tradition of both storing knowledge and
communicating decisions. At the same time, documentation is
a tedious, time-consuming task that is usually reduced to a
minimum in continuous software development processes
such as lean, agile and DevOps. Continuous software development
has been discussed in. The focus of this mapping study is
on documentation practices in continuous software development
processes such as lean, agile and DevOps. These development
processes are the de facto standards in many small startups as
well as in large enterprises. A mapping study for documentation
in continuous software development processes does not exist.
Because documentation in these processes deviates from textbook
standards that are taught during education, and there is no
prescribed standard but just a practice of documentation,
this study is relevant for both researchers, practitioners,
and educators. This mapping study is an assessment of
existing literature on development processes, documentation
methods, and frameworks -including tools. It aims to find
and classify the primary studies in this specific topic area.

RESEARCH QUESTIONS
RQ1: What studies exist on documentation practices in
continuous software development (CSD)?
Rationale: Documentation plays a major role in preserving
knowledge and communicating decisions in software
architecture and technical implementation. At the same time,
documentation practices in CSD are lacking. With this
research question, an overview of documentation methods
will be presented.

RQ2: What studies exist on tools used in CSD?
Rationale: In the community of practice for continuous
software development, tools are used to speed up
development, monitor quality, and automatic deployment.
This documentation is not exported to a central repository
but kept with the tool, e.g. Jira, GitHub. The focus
is primarily on tools that are described in the literature
but will be extended to tools that are actually used
for architects and developers.
```

1. *A Study of Enabling Factors for Rapid Fielding: Combined Practices to Balance Speed and Stability*,

Stephany Bellomo and Robert L. Nord and Ipek Ozkaya, 2013

2. *A Study of the Documentation Essential to Software Maintenance*, Sergio Cozzetti B. de Souza and Nicolas Anquetil and Káthia M. de Oliveira, 2005
3. *Agile Architecture Interactions*, J. Madison, 2010
4. *Agile Architecture IS Possible You First Have to Believe!*, M. Isham, 2008
5. *Agile Documentation, Anyone?*, B. Selic, 2009
6. *Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects*, Rüping, Andreas, 2005
7. *Agile in Distress: Architecture to the Rescue*, Robert L. NordIpek OzkayaPhilippe Kruchten, 2014
8. *Agile software development: the business of innovation*, J. Highsmith; A. Cockburn, 2001
9. *Agility and Architecture: Can They Coexist?*, P. Abrahamsson; M. A. Babar; P. Kruchten, 2010
10. *Analysis and Management of Architectural Dependencies in Iterative Release Planning*, Brown, Nanette and Nord, Robert L. and Ozkaya, Ipek and Pais, Manuel, 2011
11. *Architecting for Large-Scale Agile Development: A Risk-Driven Approach*, Ipek Ozkaya, Michael J. Gagliardi, Robert Nord, 2013
12. *Architecting for sustainable software delivery*, Koontz, Ronald J and Nord, Robert L, 2012
13. *Architecting in a Complex World: Eliciting and Specifying Quality Attribute Requirements*, Wojcik, Rob, 2013
14. *Architects as Service Providers*, R. Faber, 2010
15. *Beyond Scrum + XP: Agile Architecture Practice*, Ozkaya. Ipek, Robert L. Nord, Stephany Bellomo, and Heidi Brayer , 2013
16. *Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development Towards Continuous Deployment of Software*, Olsson, Helena Holmstrom and Alahyari, Hiva and Bosch, Jan, 2012
17. *Combining architecture-centric engineering with the team software process*, Nord, Robert L and McHale, James and Bachmann, Felix, 2010
18. *DevOps: A Software Architect's Perspective*, Bass, Len and Weber, Ingo and Zhu, Liming, 2015
19. *Elaboration on an integrated architecture and requirement practice: Prototyping with quality attribute focus*, S. Bellomo; R. L. Nord; I. Ozkaya, 2013
20. *Enabling agility through architecture*, Brown, Nanette and Nord, Robert and Ozkaya, Ipek, 2010
21. *Enabling Incremental Iterative Development at Scale: Quality Attribute Refinement and Allocation in Practice* , , 2015
22. *Evolutionary Improvements of Cross-Cutting Concerns: Performance in Practice*, Bellomo, Stephany and Ernst, Neil and Nord, Robert L and Ozkaya, Ipek, 2014
23. *Integrate End to End Early and Often*, Bachmann, Felix H and Carballo, Luis and McHale, James and Nord, Robert L, 2013
24. *Making Architecture Visible to Improve Flow Management in Lean Software Development*, R. L. Nord; I. Ozkaya; R. S. Sangwan, 2012
25. *Microservices Architecture Enables DevOps Migration to a Cloud-Native Architecture*, Balalaie, Armin and Heydarnoori, Abbas and Jamshidi, Pooyan, 2016
26. *Microservices tenets*, Olaf Zimmermann, 2017
27. *Migrating to Cloud-Native Architectures Using Microservices: An Experience Report*, Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi, 2015
28. *Peaceful Coexistence: Agile Developer Perspectives on Software Architecture*, D. Falessi; G. Cantone; S. A. Sarcia'; G. Calavaro; P. Subiaco; C. D'Amore, 2010
29. *Presenting a framework for agile enterprise architecture*, B. D. Rouhani; H. Shirazi; A. F. Nezhad; S. Kharazmi, 2008
30. *Software Architecture for Developers Technical leadership by coding, coaching, collaboration, architecture sketching and just enough up front design*, Brown, Simon, 2014
31. *Software Specification and Documentation in Continuous Software Development: A Focus Group Report*, U. Van Heesch and T. Theunissen and O. Zimmermann and U. Zdun, 2017
32. *Sustainable Architectural Design Decisions*, Zdun, Uwe and Capilla, Rafael and Tran, Huy and Zimmermann, Olaf, 2013
33. *The DevOps Handbook : How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, Kim, Gene; Humble, Jez; Debois, Patrick; Willis, John, 2016
34. *The impact of agile practices on communication in software development*, M. PikkarainenJ. HaikaraO. SaloP. AbrahamssonJ. Still, 2008
35. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, Behr, Kevin; Kim,

Gene; Spafford, George, 2014

36. *Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail*, S. Bellomo; N. Ernst; R. Nord; R. Kazman, 2014
37. *Understanding the Role of Constraints on Architecturally Significant Requirements*, Neil Ernst, Ipek Ozkaya, Robert Nord, Julien Delange, Stephany Bellomo, Ian Gorton, 2013
38. *Variations on Using Propagation Cost to Measure Architecture Modifiability Properties*, Nord, Robert L. and Ozkaya, Ipek and Sangwan, Raghvinder S. and Delange, Julien and González, Marco and Kruchten, Philippe, 2013
39. *Working Together: The Team Software Process and Architecture-Centric Engineering*, , 2013

Chapter 6

In Continuous Software Development, Tools Are the Message for Documentation[1]

Abstract In Continuous Software Development, a wide range of tools are used for all steps in the life cycle of a software product. Information about the software product is distributed across all those tools and not stored in a central repository. To better understand the software products, the following media elements must be taken into account: the types of information, the tools, tool-stacks and ecosystems to manage the (types of) information, and the amount of structure. In the tile, "tools" refers to the phrase "the medium is the message", coined by McLuhan and Fiore (1967) pointing that the medium should be subject of investigation as well as the content of the message. In this paper the tools include tool stacks, ecosystems, the types of information and amount of structure; they define the content of the message. Our approach to present relevant information to different stakeholders is rooted in understanding and utilizing these aspects. In this respect, the amount of structural variety of information defines the value for information creation and retrieval, including the tools to process that information. Documentation is considered an information type that is processed through tools in a software development ecosystem.

Keywords Agile, Continuous Software Development, DevOps, Documentation, Lean

6.1 Introduction

In traditional software development, the main tools for developers are limited to a small number of tools such as an IDE and Source Control Management (SCM). In modern software development approaches such as Lean, Agile, and DevOps, an increase can be observed in the overall number of tools developers are using. There are many tools for project management, ranging from simple task management tools like Trello, to enterprise project management with Jira, or a wide range of IDEs such as VSCode, Eclipse, or IntelliJ. Even categories used for classifying tools are manifold, ranging from data management to development tools and from deployment tools to monitoring tools [S97], [213]. Information about software is scattered throughout all the tools used in a software development ecosystem. For most software products, there is no single repository

[1] This work was originally published as:

T. Theunissen, S. Hoppenbrouwers, and S. Overbeek, "In Continuous Software Development, Tools Are the Message for Documentation," in *Proceedings of the 23th International Conference on Enterprise Information Systems*, 2021. doi: 10.5220/0010367901530164.

that contains all information about the software. For instance, the core concept of a software product may be presented in PowerPoint-like tools. Information about stakeholder concerns, risks, constraints, and context may be documented in Word-like documents, modifications on source code are often maintained in git, and deployment documentation may be defined as executable infrastructure-as-code. These examples show that the type of information documented has a strong relationship with the tool it is stored in and used with. These tools often define the format of the information. The format can range from structured text to video. Documentation in modern software development concerns the creation of information about the software product, conveying knowledge about the software product, and in some cases even executing the documentation. The scattered information that is stored in the myriad of tools introduces issues of retrieval and comprehension of relevant information about the software with respect to the stakeholders involved. The phrase "tools are the message" concerns:

1. the **types** of information documented,
2. the amount of structural **variety**: whether the information is structured (source code, templates) or unstructured (sketches, text),
3. the myriad of **tools** used, including tool categories organized into stacks and ecosystems, and
4. **comprehensibility** support of the software product.

To better understand why tools are the message, we introduce the umbrella term "CSD" that covers the characteristics of Lean, Agile, and DevOps software development approaches. First, it covers the values, principles, practices, processes, and tools for Lean, Agile, and DevOps. Second, CSD embraces the whole life cycle of software as a product, from conception to end-of-life. This includes continuous design and architecting, and also development until retirement. Third, CSD takes into account the drivers behind the continuously changing state of the software product, such as progressive insights, contextual changes, new features, bug fixes, or other unforeseen factors. Last, information about the software is distributed across the many tools used in a software development ecosystem. "The tools are the message" will be explained by showing how the types of information documented relate to the tools used for information creation, retrieval, and execution.

The scientific contribution of this paper is the insight that the requirements for documentation in lean, agile and DevOps are present in values of these methods, as well as in the community of practice of industrial software engineers. We present the generic requirements and conditions for documenting and communicating contained in CSD knowledge. Our findings lead to approaches for knowledge preservation in CSD.

In the remainder of this paper, the following subjects will be addressed. In Section 6, the study design is presented. In Section 6 the data collection, data analysis and data interpretation are shown. In Section 6, the types of information are discussed. "Tools are the message" is discussed in Section 6. The paper ends with conclusions in Section 6.

6.2 Study Design

In this study, we use multiple sources for data collection and data analysis to enhance its credibility. We will use Multivocal Literature Review (MLR) following Garousi, Felderer, and Mäntylä [214] and we follow Yin [215] for the case study design.

6.2.1 Multivocal Literature Review (MLR)

The motivation for conducting an MLR is that the investigation of documentation in CSD is relatively new. Many state-of-the-art values, principles, knowledge, practices, tools and processes are shared in, for example, blogs, (online) lectures, data sets, and (technical) reports. Therefore, it is important to take these sources into account besides the available body of literature on the topic [214].

6.2.1.1 Quality Assessment Criteria for Data Sources

In Table 6.1, the columns with 'grey' and 'black' literature types are not ordered by rigour. Added to this list are git software repositories, as they contain information on documentation about software. The selected repositories, displayed in Table 6.2, are open source with a hundred to a thousand contributors. They deliver high quality software that has often been running for years.

6.2.1.2 Inclusion and Exclusion Criteria

Search engines used for the MLR, based on popularity [221] and triangulation, are:

'White' literature	'Grey' literature	'Black' literature
• Published journal papers • Conference proceedings • Peer-reviewed books	• Preprints • Technical reports • Lectures • Data sets • Podcasts, Video • Blogs • Git SCM	• Ideas • Concepts • Thoughts

Table 6.1: Types of literature, based on Garousi, Felderer, and Mäntylä [214].

ID	Source	Type	Description	RQ1	RQ2	RQ3	RQ4
S1	bash [216]	SCM	Open source project for a Linux shell		✓		✓
S2	Mozilla open source project [217]	SCM	Open source project for Firefox, a web browser		✓		✓
S3	LATEX open source project [218]	SCM	Open source project for a document editing tool		✓		✓
S4a	stackshare.io [213]	Data set	Website that collects user generated data on popular development and deployment stacks	✓			✓
S4b	thoughtworks.com/radar [219]	Data set	An opinionated guide to technology frontiers	✓			✓
S4c	gartner.com [220]	Data set	Interpreting technology hype	✓			✓
S5	Case study	Data set	Interviews with practitioners from the industry	✓	✓	✓	✓
S6	Documents	Data set	Documents that support the interviews	✓	✓	✓	✓

Table 6.2: Description of Data sources and Types used in this paper and contribution to answering the research questions.

- Google. 70.0% popularity on Desktop, 94.0% on Mobile devices.
- Bing. 13.2% popularity on Desktop, 0.7% on Mobile devices.
- DuckDuckGo (DDG). 0.3% on Desktop, 0.2% on Mobile devices.
- Google Scholar (GS). Not mentioned by NetApplications [221]. It was used for triangulation to compare with academic search engines.

Google Scholar was added to verify the results from the other search engines. Furthermore, if Google Scholar returns few search results, then search engines like ACM, IEEE, ScienceDirect, Springer or WebOfScience often also show few results. Other search engines mentioned by NetApplications [221] are powered by Google (AOL) or Bing (Yahoo) and therefore considered redundant. Only western search engines were included; Asian search engines were excluded.

Garousi, Felderer, and Mäntylä [214] refers to the extensive assessment of sources, compared to 'white literature'. In Table 6.4, the inclusion criteria are presented. For definition of the numbers, the threshold is randomly selected, or top ranking is applied. The numbers for the thresholds are set to cover the research questions.

ID	Criteria
C1	Open source projects, including number of developers. The threshold is 10 developers.
C2	Open source projects, including number of years existing. The threshold is 10 years.
C3	Open source projects, including format of documentation. Range of formats over projects are selected to cover the variety of information.
C4	Community generated data data sets including metrics for stacks. The top three are taken into account.
C5	Community generated data data sets including metrics for companies.The top three are taken into account.
C6	Community generated data data sets including metrics for developers. The top three are taken into account.

Table 6.3: Inclusion Criteria, following Garousi, Felderer, and Mäntylä [214] and Kitchenham and Charters [222].

In Table 6.4, C3 concerning the formats, refers to the following formats: HyperText Markup Language (HTML), Compiled HTML Help (CHM), Rich Text Format (RTF), Portable Document Format (PDF), LATEX, PostScript, man pages, DocBook, Extended Markup Language (XML), and ePub.

6.2.1.3 Search Process

The search terms on all three search engines are: "`stacks tools technology`". In the test run, it became clear that the search results for Google Scholar were useless, unless the term 'software' was added. The final search string was "`stacks tools technology software`". Each time, an anonymous browser session was started to minimize hints from the search engines based on previous searches. Different search engines use different metrics to calculate the result set. The number designates an indication, not a rank.

Search Engine	Number of hits	Top three results
Google	47.900.000	Stackshare homepage • Popular Tech Stacks from stackshare • Technology Stack: What it is and how to build one on mixpanel.com
Bing	141.000.000	Popular Tech Stacks from stackshare • Stackshare homepage • https://mopinion.com/tools-for-your-2019-marketing-technology-stack/
Duck Duck Go	Not calculated	Popular Tech Stacks from stackshare • Stackshare homepage • Top six stacks from fingent.com
Google Scholar	289.000	"What are developers talking about?" from Springer • "Singularity: rethinking the software stack" from ACM • "Managing the Life Cycle of Linked Data with the LOD2 Stack" from Springer

Table 6.4: Inclusion Criteria, based on Garousi, Felderer, and Mäntylä [214].

6.2.2 Case Study

We follow Yin [215] for the case study approach. A case study is appropriate for research that answers "why" and "how" questions. This study does not require control of behavioral events, and focuses on contemporary phenomena [215].

The following types of research are derived from Yin [215].

- Explanatory
  Description of cause-effect relations used with inductive reasoning and often used with descriptive statics. This type of research is applicable because, for understanding relations between real-life phenomena, the interviewees can elaborate on the phenomenon and their relation. Typical questions start with 'who', 'what', 'where', 'how many' and 'how much'.
- Descriptive
  Takes the context into account by scientifically reporting observations about situations and events. In real-life situations, situations and events can not be considered without their context. Typical questions start with 'why' and 'how'.
- Exploratory
  Defines and tests hypotheses for building new theories. For this paper, exploring and building new theories is not the primary objective, but the collected data can serve in follow-up research. Typical questions starts with 'why' and 'how'.

Figure 6.1 depicts the case design and units of analysis.

6.2.2.1 Units of Analysis

The units of analysis are individual practitioners from (non-)profit national and international organizations in senior positions. Professionals in IT-industry are relevant because IT is the domain of research. The reason for doing individual interviews is that they can provide in-depth information, and individual or even opinionated perspectives on matters of concern. Also, for practical reasons, it is easier to arrange a meeting with individuals than with groups. Concerning seniority, practitioners have an overview of the continuity of software projects over the years, including software products' evolution. This includes changing the technology stack, organizational change, and the IT tooling landscape. There is no particular reason for including national or international organizations. Practical reasons such as availability are decisive in selecting organizations. The teams' and organizations' size is relevant because knowledge about the software product is more present in larger teams, including historical knowledge about decisions, bugs, and bug fixes.

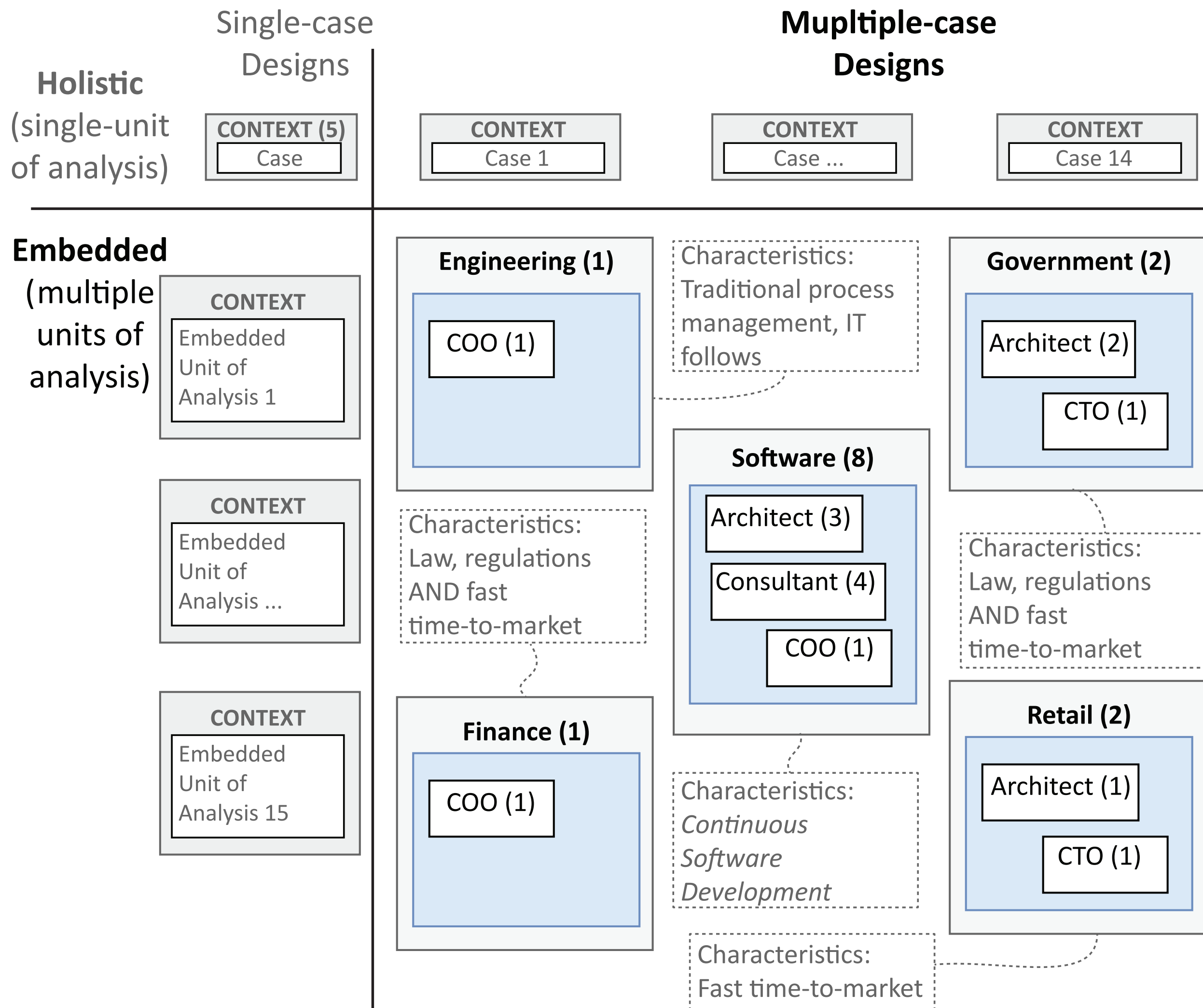


Figure 6.1: Units of analysis -individual practitioners- with cases from multiple types of organizations, including multiple departments, concerning documentation in CSD.

6.2.3 Objectives and Research Questions

The *objective* of this study is defined in the main research question:

> *The objective is to investigate the necessary and sufficient conditions to organize information scattered throughout a CSD ecosystem into comprehensible documentation for designated stakeholders.*

The related research questions are:
- RQ1: **Which tools** are used in the software development ecosystem?
- RQ2: What is the **variety** of information that is stored in tools?
- RQ3: Which information is stored with what tool?
- RQ4: How can this scattered information be organized into **comprehensible documentation**?

6.3 Results

In this section, the data collection methods, data analysis methods and data interpretation methods are presented, together with the data.

6.3.1 Data Collection

Table 6.2 presents the sources and types for the data used in this paper. From a wide range of popular open source projects, S1-S3 were selected because of their contribution to answering RQ2 concerning variety of information and RQ4 concerning organizing information into comprehensible documentation. Sources S4-S6 contribute to answering RQ1, namely which tools are used in which ecosystem. From the publicly available data sets that present the popularity of tool stacks, stackshare.io was the only viable option. Alternatives, such as stack.g2.com or www.producthunt.com show limited sets. Sites such as alternativeto.net show only alternatives for a specific tool. The data from stackshare was compared with information collected from the interviews. Sources S5-S6 contribute to answering all research questions and were selected to validate and extend the data collected from S1-S4.

6.3.1.1 Data Collected from the Source Code

The source code from the open source project was selected because repositories with a long development period also have numerous developers. New developers build on existing code and sometimes need to work through a wide range of standards and guidelines, architecture, decisions, UI, or enforced coding standards by the use of code linters (a linter is a static analysis tool that warns for or prohibits deployment when code is not following styles or constructs, or includes programming errors). Examples are mediawiki and Linux.

The repositories presented in Table 6.2 are selected because they differ in the **type of medium** for documentation. This contributes to answering RQ2.

6.3.1.2 Data Collected from the Data Sets

Three sources are used: S4-S6 in Table 6.2. The tools from stackshare.io are used to get an overview of popular tools and tool stacks. The data retrieved from the interviewees verifies the popularity of tools in tool stacks. Furthermore, the number of tools in tool stacks mentioned by interviewees is higher and gives insight into the selection of tools.

6.3.1.3 Interviews

We held 14 interviews (S5 in Table 6.2) with 15 subjects in 5 different cases. The cases were selected based on the expected motivation for documentation, ranging from fast TTM to following regulations, as mentioned by Bass, Weber, and Zhu [3]. Figure 6.1 depicts the units of analysis (interviewees) with multiple cases (organizations with different motivations for documentation).

6.3.2 Data Analysis

In this section, filtering, grouping, ordering, and visualization of data is presented. The filtering applies to a selection of data that falls within this study's scope and contributes to answering the research questions. Discussions and insights in the interviews that do not contribute are not taken into account. The grouping of the data applies to combine results in aggregated classes with common properties. Part of the grouping matches with the subsequent research questions. The ordering refers to the relevance of visual properties. With the visualization, a quick and comprehensive overview is presented from relevant data.

In Figure 6.2, the types of organization are represented. Based on Bass, Weber, and Zhu [3], it was expected that the motivation for documentation ranges from fast TTM, such as in retail, to required documentation because of regulations, such as in government or finance. Software companies are defined having IT as their core competence, and having more than 80% of their revenue generated by IT. Modern companies use their digital infrastructure platform as their source of revenue, but none of the interviewed companies generates revenue from a platform.

In Figure 6.3, the functions of the interviewees are represented. All 15 interviewees have senior positions, either as some kind of manager or some kind of technician. Consultants are technical consultants, not business consultants although the senior consultants have a wider span of control and skills than IT only. C-level interviewees all had an education in IT and were seasoned IT practitioners.

Figure 6.4 presents on the x-axis the number of tools in a tool stack, based on numbers from stackshare.io. The left y-axis represents the number of found tool stacks with the number of tools. Combinations most found numbers three. There are 95567 combinations of three tools (blue line). The red line depicts the number of

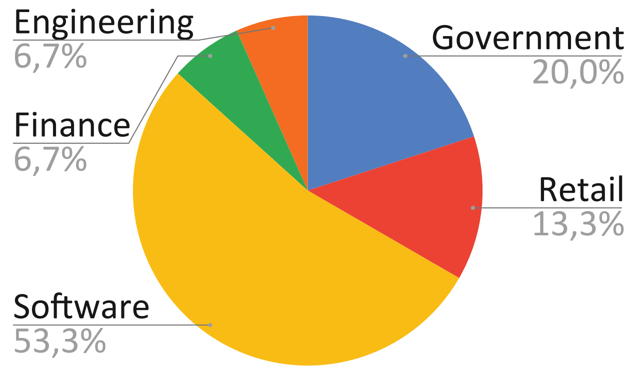


Figure 6.2: Types of organization from the interviewees.

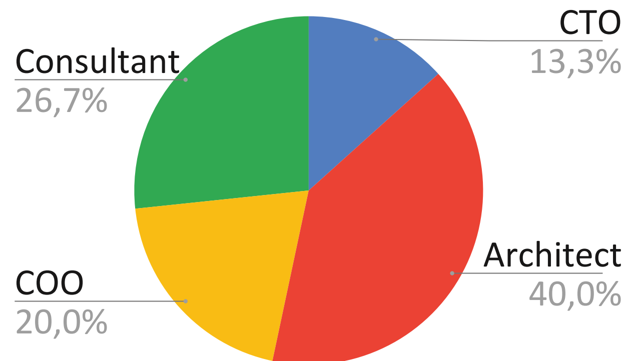


Figure 6.3: Functions of interviewees.

most popular combinations. A further analysis of the data is in the Appendix[2]. The table represents popular combinations and the number of combinations. The most popular combination is GitHub with nginx and Redis. This combination is mentioned 47 times by the stackshare.io. If the combination of tools is 11 or more, there are fewer combinations of tools in tool stacks. Another interesting observation is that productivity tools are more popular than communication or documentation tools. The first appearance of a communication tool is the chat application Slack, with a number of 5 tools or more. Trello, as task management tool, is mentioned when a tool stack is built up of 8 tools or more. The first time a documentation-like tool (Markdown) is mentioned is when a tool stack consists of 13 tools or more.

Based on the interviews, the number of tools was higher than the popularity index in Figure 6.4. Interviewees mentioned a full range of tools that are either prescribed and supported or tools that are allowed for individual developers. An example of this is that the tool stack from JetBrains is prescribed and supported but developers have the option to use another IDE (VSCode). Another example is Postman for the testing of REST API endpoints. This tool is allowed, can be highly productive for individual developers but is not supported by the company. Communication and collaboration tools are not allowed to be individual choices but are prescribed by the company.

Figures 6.5 and 6.6 were initially compiled from Theunissen, van Heesch, and Avgeriou [20]. The figures were presented in the interviews for validation and extended with other types of information.

6.3.3 Data Interpretation

With data interpretation, the research questions are answered based on the analyzed data. Also, deviations from expectations are discussed.

One deviation from the expectation is the motivation for documentation, which affects multiple cases in the design. Bass, Weber, and Zhu [3] mention a range of motivations between fast TTM for retail to following

[2]Appendix on https://theotheunissen.nl/tools-are-the-message

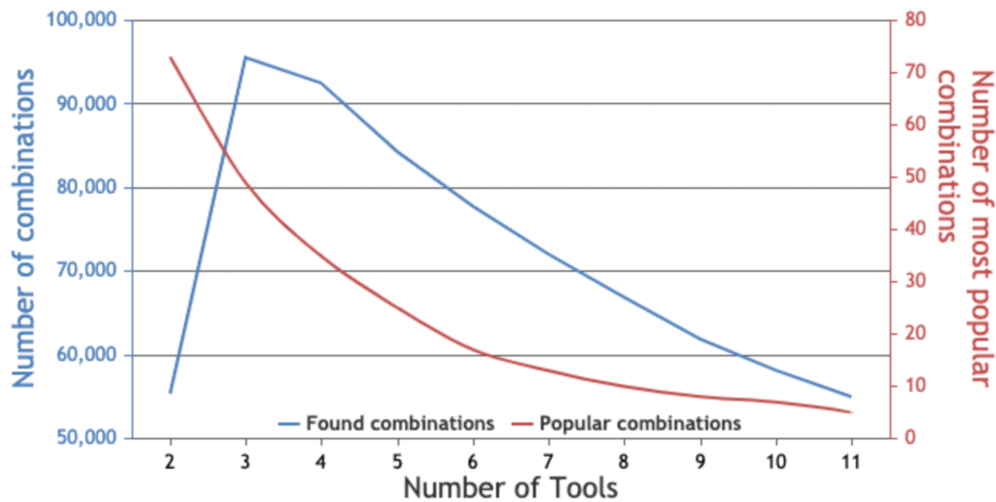


Figure 6.4: Tools and tool combinations, based on stackshare.io and theotheunissen.nl/tools-are-the-message.

regulations for government. This range defines the multiple cases in the design. However, although government and finance have to apply to strict regulations, the need for fast TTM is immanent. For governmental organizations in the Netherlands, national elections held at least every four years result in new policies that have to be implemented at short notice. For governmental organizations, no trade-off is possible. These organizations have to comply to law and regulations, and follow policies. The workaround is to automate development as much as possible and fill the gaps with manual operations. Financial companies also have to conform to regulations and standards such as AMLD5[3], GDPR[4] or the Gramm-Leach-Bliley Act [5]. The notion of 'fast TTM' applies to keeping up with regulations. Fast TTM does not refer to adding up-selling and cross-selling features in a web shop for the financial industry. The effect on the multiple cases in the case study design is that the cases for governmental organizations and the financial industry are identical and not different cases. However, this does not affect the answering of the research questions.

A second observation is that interviewees mention many more tools than are mentioned on stackshare. The sources contributing to answering RQ1 are the online user generated tool stacks with tools from stackshare.io (S4a). These results where verified, validated, and extended by the interviews (S5) and supporting documents (S6). Figure 6.4 show that the most popular sets consist of three tools. There are 95567 stacks with three tools. However, 69262 stacks are only mentioned once with three tools, making the number of popular stacks a long tail. Interviewees mentioned over 20 tools. This deviation has a small effect on RQ1 -which tools are used- because the range of tools is retrieved and confirmed with interviewees. For RQ3, which type of information with what tool, it has a larger effect because interviewees could elaborate on the motivation. Stackshare.io only shares numbers, not motivations. The interviewees all mentioned more tools than could be extracted from stackshare and the tools from stackshare are also mentioned by interviewees. The effect from the interviewees is only qualitative because they can motivate choices, describe relations between tools, or refer to organizational policies.

6.4 Types of Information

In this section, the types of information in CSD are presented. There is a distinction between information and documentation. The term **information** is used to refer to any (set of) symbols that

1. makes a difference [5], [223] and
2. causes one or more effects [6].

The term **documentation** is used for any written, verbal, visual artifact or activity that transfers knowledge between stakeholders, related to the software product [S145]. Documentation stems from the etymological meaning for [4]:

[3]https://eur-lex.europa.eu
[4]https://gdpr-info.eu
[5]https://ftc.gov

1. teaching (Latin: *docere*),
2. pointing out, or
3. instructing with evidence and authority.

The types of information are the distinctive properties for what is actually documented, at what point in the process in CSD, how it is stored, and why it is relevant to keep the information. Figure 6.5 presents the types of information, tool categories, and examples of tools.

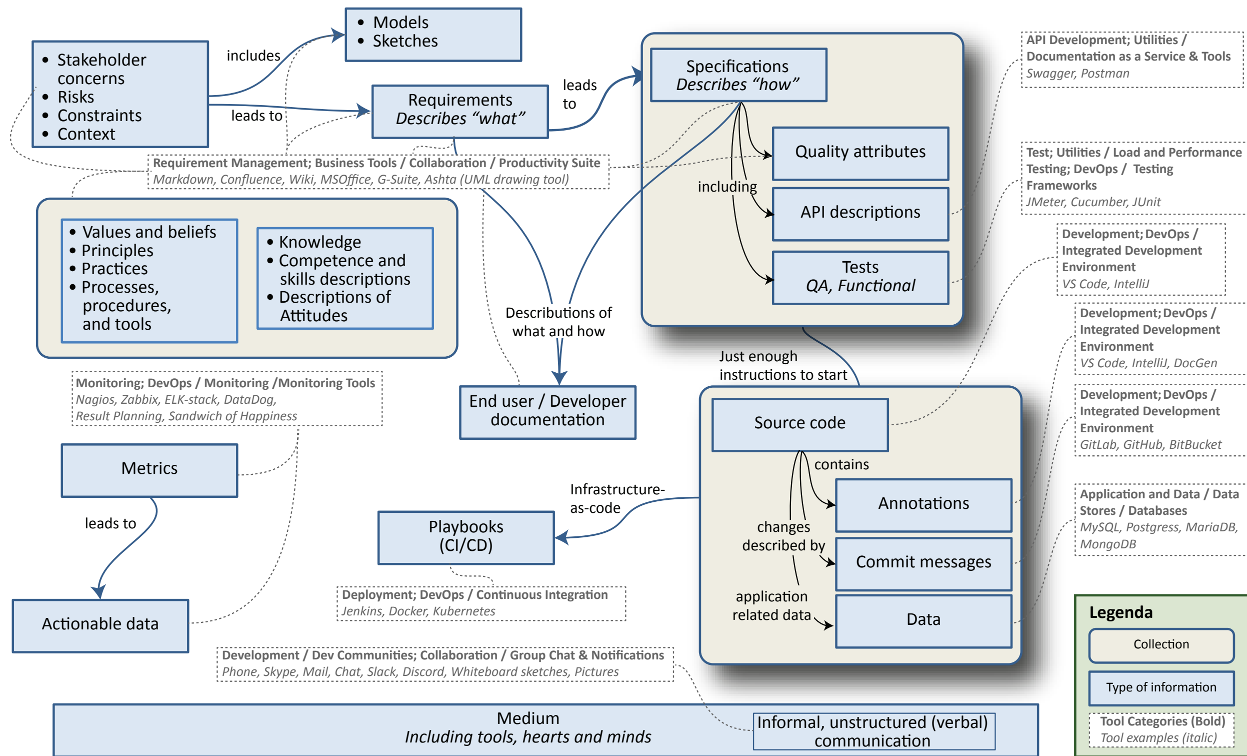


Figure 6.5: Types of Information, including mapping to tool categories and tools. The medium applies to the communication of information. Metrics and actionable data apply to all information for insight and control of processes.

6.5 Tools Are the Message

Marshall McLuhan coined the phrase "the medium is the message" [224]. He pointed out that the medium should be subject of investigation as well as the content of the message. The subject for this study concerns both the tools as well as the content of the message, so note that tools are part of the message and not the only message. "The tools are the message" thus refers to a different message being communicated if tools make use of different media types such as written, verbal, or visual. The primary concern for this study is the tools, not the types of information. However, the types of information do have a strong relationship with the tool in which the information is created, retrieved or updated. In this section, the aspects that establish that tools are (part of) the message will be discussed. In Section 6 the tools, tool-stacks and software development ecosystems are discussed. Tools define which tools, stacks or ecosystems are used when information is created, captured, understood or processed. In Section 6 the variety of information is discussed. The variety refers to the amount of structure information has. This amount of structure is defined by creation and retrieval of information with tools. In Section 6, the relation between the types of information, tool (stacks), and the variety of information will be presented as "tools are the message".

6.5.1 Tools, Tool Stacks, and the Software Development Ecosystem

In this section, the tools in relation to other tools are described. The tools, tool stacks and their popularity can be found in Sources S4 (websites), S5 (interviews), and S6 (supporting documents to the interviews), displayed

in Table 6.2.

In CSD, tools are organized into stacks, and tool stacks are organized into software development ecosystems. In Figure 6.6, an overview is presented for the relation between Software Development Ecosystems with Tool

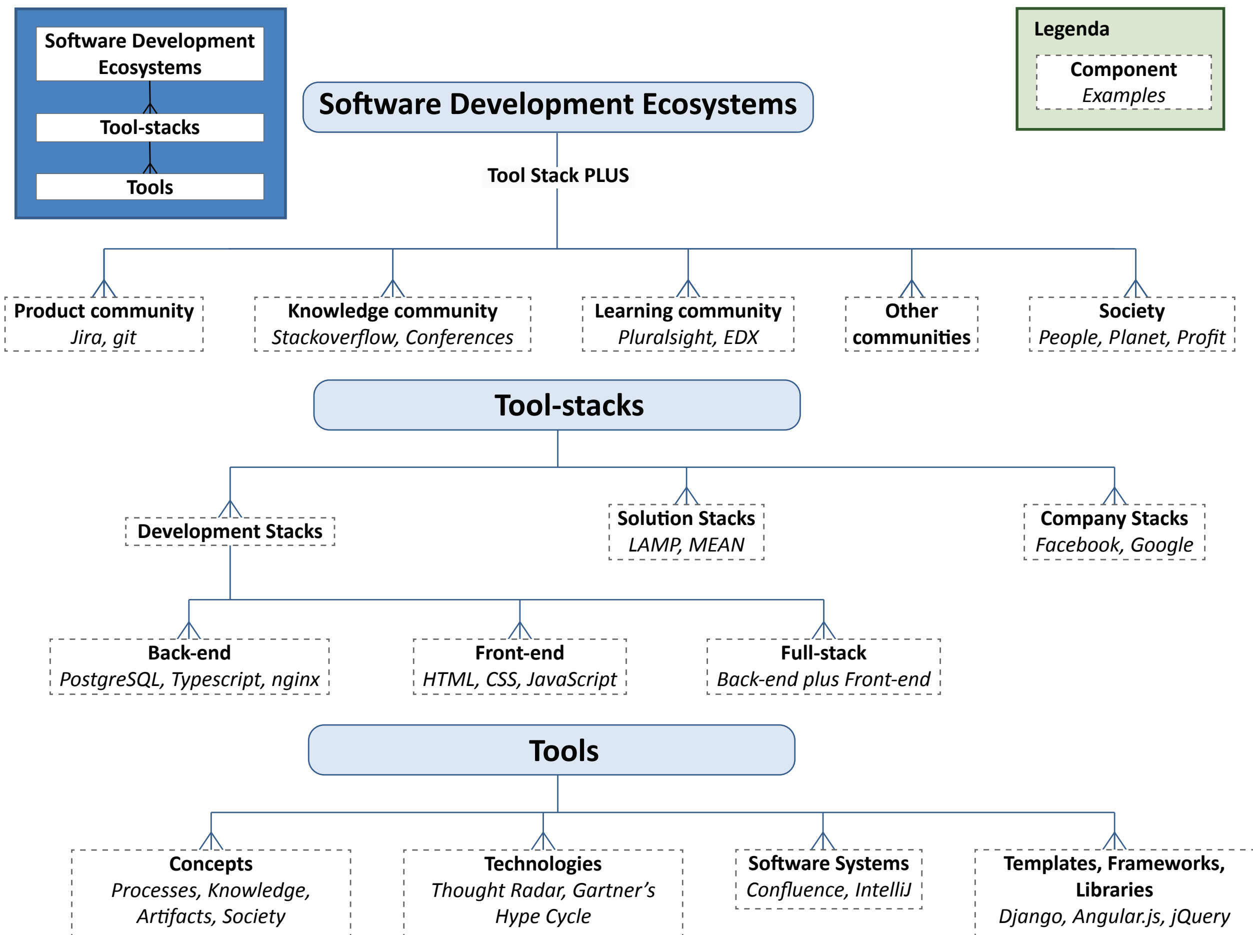


Figure 6.6: Relations between Software Development Ecosystems with Tool stacks and Tools, including components and examples.

stacks and Tools, including components and examples. In the next paragraphs, an explanation will be presented of the figure.

1. **Tools** A tool is defined as a concept, technology, software system, template, framework, or library to design, develop, and maintain a software product. The level of freedom can define the difference between a template, framework, and library. A (technology) template is like a form where the only freedom exists in filling in the value of variables, such as a Python Django template. A framework is a comprehensive set of methods that prescribe the purpose and usage of functions and methods. The maximum freedom is with libraries that provide a set of functions that can be used to speed up development, e.g., jQuery. Examples for concepts are the nature of technology, such as the processes, knowledge, and artifacts, including impact on society [225]. For technologies: Technology Radar or Gartners Hypecycle. Examples for software systems are typical tools like Confluence and IntelliJ.
2. **Tool stacks** The organization of tools into tool stacks for 1) development stacks including back-end, front-end or full-stack, 2) solutions stacks, or 3) company stacks. The *front end* is where the end-user interacts with the system. For web applications, a limited set of technologies is available. For mobile application development, there is no dedicated technology to create the user interface, other than supporting tools to create wireframes or graphic tools to create low-fidelity or high-fidelity User Interface (UIMLR)s. The stack for *back end* development concerns databases, applications, and servers. This also includes the infrastructure the software is running on. This might be on-premise, in the cloud, or hybrid. A myriad of tools for the back end is available. At stackshare, over 2,000 tools can be found related to back-end

development [213][6]. The *full stack* can be viewed as the sum of tools for the back end and front end. However, this simple sum of tools also includes processes, knowledge, and competences, and other types of information and not just the tools but also to understand and manage the complexity of this sum of tools. This fits with the processes in DevOps, where developers and maintainers are the same people. A *solution stack* is a specific set of tools to solve a problem where each tool contributes to the solution, and tools are mutually exclusive. Tools can be exchangeable, for instance MySQL with MariaDB, or Apache with nginx. A *company stack* is a stack of tools used by a company for its specific situation, focusing on fast TTM or following legislation. In this respect, the reason for documentation is relevant. It can lead to very little documentation in case of fast TTM or to mandatory documentation because of regulatory reasons, as in the aviation industry, medicine industry, or tax administration.

Examples for 1) are github and intelliJ. Examples for 2), the solution stack, are Linux, Apache, MySQL, PHP (LAMP), Mongo, Express, Angular, Nodejs (MEAN). For 3), the company stacks, examples are for Facebook: Hack, React, Cassandra, and GraphQL or Google with Dart, Go, Angular.js, and Material Design. The stack as a concept is discussed by Jansen, Finkelstein, and Brinkkemper [226], referring to Software Ecosystems (SECO). This is a set of business functioning as a unit interacting with a shared market for software and services, including relationships. SECO already points to a context, but the community takes the context explicitly into consideration as is discussed in the next paragraph.

3. **Software Development Ecosystems** This is an extension of the tool stacks with communities, optionally outside the team or company. A *product community* refers to information about specific tools, including concepts, technologies, software systems, and templates, frameworks, and libraries. The community for *knowledge* can be based either on websites where developers gather to exchange questions, answers, and contemplation, or on conferences where developers from industry and scientists meet. There are also meet-ups specific for industry developers, that are also attended by scientists. This makes clear that knowledge sharing involves, besides cognitive activity, also social activity. The *learning community* includes sites for Masssive Open Online Course (MOOC)s, either academic or commercial, to learn about concepts and practicing code, but also classic learning environments for students at universities.

   Examples are product or tool sites. Knowledge sharing sites like Stackoverflow, Reddit, or Quora; online courses from Pluralsight, EDX, or Udemy.

The classification of tools into tool stacks varies from an unorganized landscape such as Kersten [S97] to layers for Application and Data, Business Tools, DevOps, and Utilities such as Stackshare.io [213]. Other classifications take into account the processes, infrastructure, productivity (4+1 from Kruchten [S178], or C4 from Brown [210]).

The way the tools are organized into tool stacks, and tool stacks are organized into software development systems, shows that this "scattering" has a high level of organization. The tools are not randomly picked to serve a purpose, but are combined to support design, development, and maintenance for a software product with specific requirements from the industry such as fast TTM or regulations for documentation.

"Tools are the message" concerning the tools, tool stacks and software development ecosystems relates to the type of information that is stored with each tool. The tools in "tools are the message" define how the information is created, stored, retrieved, and communicated. The tool stack is comprised of a set of tools in use by a development team. The software development ecosystem includes the community around individual tools.

6.5.2 Variety of Information and Contribution to Knowledge Transfer

In this section, the variety of information is discussed, including the relation with tools. This Section is closely related to RQ2: the variety of information. The data sources are S1-S3 (open source projects), S5-S6 (case studies) as represented in Table 6.2.

The information in the tools in CSD has a certain amount of structure. Figure 6.7 presents the amount of structure measured by the creation or capturing on the upper x-axis and the retrieval for human communication varying to automated processing on the right y-axis. The tools to create or retrieve the information are on the lower x-axis. The Creation dimension on the upper x-axis varies from constructing to capturing information. "Capturing" refers to the ingestion of information into a storage medium that is not created with a software development tool, e.g. whiteboard sketches, drawings, or models. A photograph might be saved in Jira or Confluence for later usage. The system does captures chat messages or email messages that have a low degree of structure. Probably the only structure an email message has is the subject and other standard headers such

[6] For this paper, the data can be found on https://theotheunissen.nl/tools-are-the-message

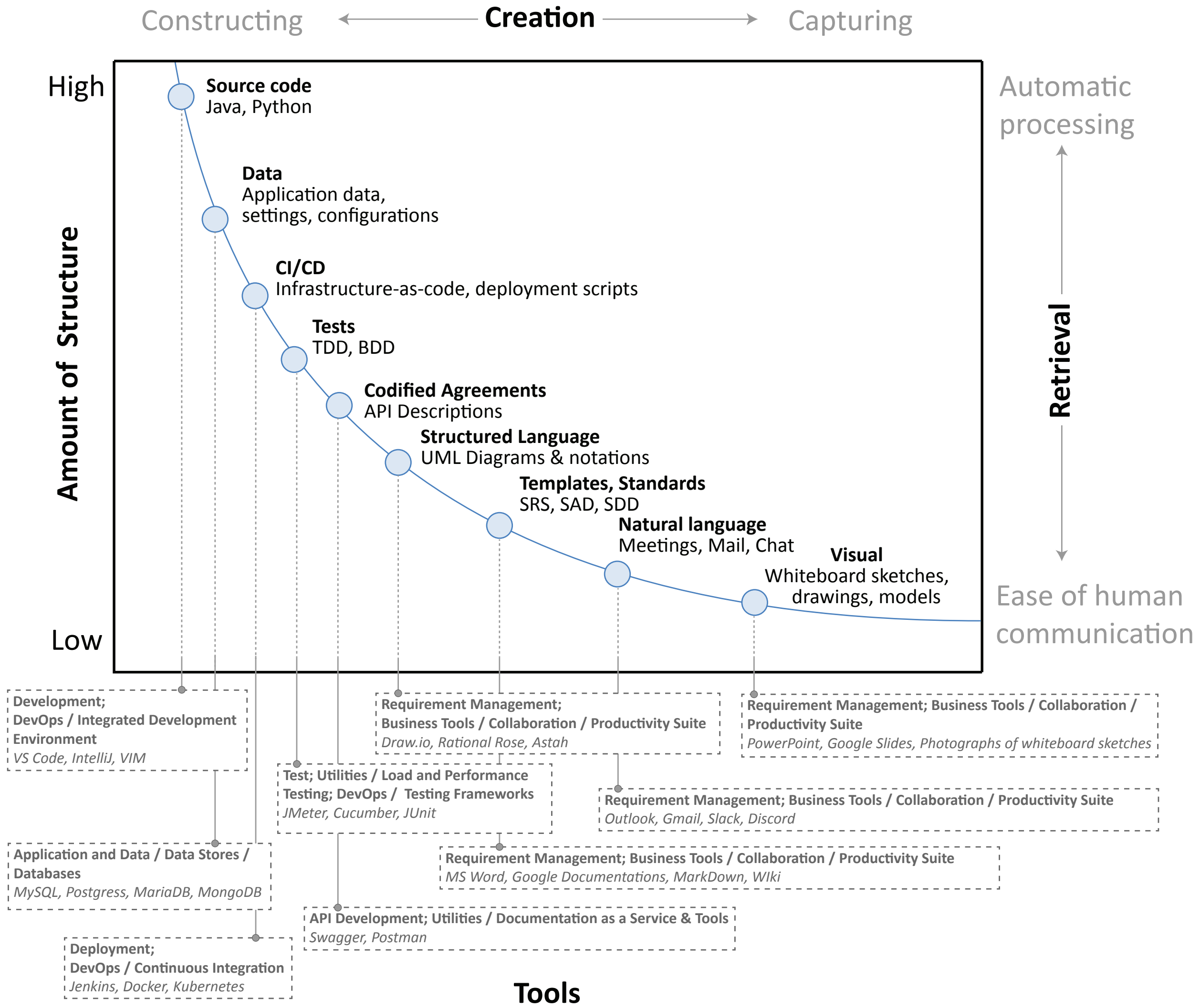


Figure 6.7: Amount of Structure measured by Creation, Retrieval, and Tools used to create or retrieve the types of information.

as the addressees. "Manufacturing" refers to the manual creation of information, such as source code. The Retrieval dimension on the right y-axis refers to the usage of the information and varies from the ease of human understanding to automatic processing. "Ease of human understanding" identifies the cognitive load required to understand the information involved. "Automatic processing" refers to the syntax, grammar, and semantics to compile source code.

"Tools are the message" concerning the variety of information relates to the amount of structural characteristics from the information that is captured, stored, or communicated. Tools enable the creation, storage, and communication of information while enforcing more or less structure.

6.5.3 Types of Information, Variety of Information, and Tools in a Software Development Ecosystem

This Section elaborates on RQ3: which information is stored in what tool. The data sources for this section are S5-S6 (interviews).

In previous sections, an overview was presented of types of information, and of tools including tool stacks and software development ecosystems. Also, the variety of information was discussed. All this makes clear that tools are significant in understanding the information and documentation that is created or retrieved. The main difference concerns the usage of the information stored in the tools, especially if the creators of the information are not the users of the information. This is most manifest when creator and user are not the same

person. Examples are teams that are geographically distributed across the world or team members that were not involved in the conceptualization of the information.

"Tools are the message" (Section 6, Figure **??**) in relationship with the types of information (Section 6), the variety of information (Section 6, Figure 6.7), and tools show that for creation and retrieval of information, tools make a difference.

For example, the presentation of a concept is better done in PowerPoint-like tools than source code only. The implementation of an algorithm can better be communicated with the actual source code accompanied with a model/sketch than in text. The creation and retrieval of information is closely related to these activities. Figure **??** presents the composition of "Tools are the message" by the types of information, variety of information and tools including tool stacks.

Considering major open source software repositories, most of the repositories show simple text documentation, including Mark Down. However, there is a close connection between the software product and the *format* of the information. For instance, for the Linux utility "bash", traditional UNIX man pages are used [216]. For LaTeX, the information about the software is in tex format [218], and for the Firefox browser, the information is in HTML-format [217]. These specific examples make clear that the software product and format of the information are strongly related.

6.5.4 How to Organize Scattered Information into Comprehensible Documentation

This section addresses RQ4. The data sources for this section are S1-S6 (all sources), as represented in Table 6.2. S1-S3 (open source projects) are used to gain insight in what (values, architecture, interfaces) is documented in what type of information (text, commit message, pictures) in what tool (Confluence, GitHub, RDBMS). S4 is used to understand the community of software practitioners. S5-S6 (case studies) are used for verification, validation and extending findings.

In CSD, "tools are the message" refers to the types of information, the variety of information and tools in software development ecosystems. The tools require or produce information with certain formats. When combining these aspects, the effort needed to comprehend the information about the software product will increase, and understanding will go down, in particular for geographically distributed teams, across buildings or across the globe, or in case of decision making not involving all team members. However, not all information about the software product is relevant for all stakeholders all the time. For customers, who pay for the development of the software product, metrics and actionable data are relevant for productivity of the software product team. For end users, a user manual should be present. If part of the end product, many websites do not have manuals but are supposed to be user friendly. For managers, metrics for relevant Key Performance Indicator (KPI)s should be present. For developers, user stories and codified API descriptions should be present.

The best way to introduce new team members to the software product is definitely not to demonstrate all tools with their variety of (un)structured information, types of information and tool (stacks). A better solution is to provide stakeholders with an overview according to their specific interest. For stakeholders, this might be a PowerPoint-like presentation with the mission and vision. For managers, this might be the metrics on a KPI dashboard. For developers, this might be the requirements, specifications, or "how to's". For end-users, this might be the user manual. Comprehensive 'yellow pages' with an overview of what is relevant for who should be available. In an investigation of the 20 major public software products, we see that the various types of information are well presented.

6.5 Conclusions

The phrase "tools are the message" is taken from McLuhan and Fiore [224] "the medium is the message". He proposed investigation of the medium instead of only the message. In this paper, the tools, including the type of information, and variety, are the "medium".

Tools are the message refers to three aspects. These aspects are:

1. the types of information
2. tools, including tool stacks and software development ecosystems
3. the amount of structure

Based on the research results, five conclusions are drawn. First, there is a *strong relationship between the type of information and the tool.* The type of information refers to the properties of the information in terms

of what is stored (content), how it is stored (format), why it is stored (relevance), and when in the process the information is stored. Tools are the concepts, technologies, software systems, and frameworks to design, develop, deploy, and maintain the software product.

The second conclusion is that *tools are organized into tool stacks, and tool stacks are organized into software development ecosystems.* A tool stack is an organized set of tools to produce a software product. The software development ecosystem includes communities outside the team, such as product communities or knowledge communities.

The third conclusion is that the *variety of tools refers to the amount of structure for information.* This amount of structure is defined by the creation and retrieval of the information, together with the tools for creation and retrieval.

The fourth conclusion is that the *combination of these three aspects makes a difference in creating, retrieving, communicating, and understanding the message.* There is a difference in communication and comprehension when understanding a software product's core concept through presenting with PowerPoint or through source code. The same applies when communicating codified agreements for the communication between subsystems through detailed endpoints, including input and output types, or through a whiteboard sketch.

The fifth conclusion is that the combination of these three aspects, including the core message, creates complexity concerning finding and understanding relevant information. However, not all stakeholders, including developers, require all information at any time. As a group of stakeholders, developers require information to start, continue, and deploy an iteration. This *focus on information from specific tools will decrease the complexity and make it easier to comprehend relevant information.*

Chapter 7

Continuous Learning with the Sandwich of Happiness and Result Planning[1]

Abstract With an increase in fast time-to-market and keeping up with fast mandatory legal changes, we observe a demand for continuous software development which is reflected by the emergence of Lean, Agile, and DevOps approaches. At the same time, we observe the phenomenon of lifelong learning that is both manifest and propagated by government, industry, and education. We introduce two patterns that match these two phenomena: the Sandwich of Happiness and Result Planning. Together, these patterns support learning for students in an educational setting and continuous learning for professionals in industry, especially in the context of Continuous Software Development.

Keywords Agile, Continuous Learning, Continuous Software Development, Documentation, Result Planning, Sandwich of Happiness

7.1 Introduction

Modern software development comes with continuous changes, reflected in agile methods and through demands for a fast time-to-market. At the same time, technology innovates in a fast pace and to keep up with these changes, lifelong learning is fundamental. In this paper we introduce two patterns to adapt to continuous change in software development by keeping up with lifelong learning. In a continuously changing environment, with new technology, engineering solutions and processes, lifelong learning is required to match these changes. The objectives of this paper are to present two patterns that can support this continuous change, and to provide insights for better comprehension and application of these patterns.

Nowadays, students as well as professionals have to frequently adapt to new situations driven by an ever-changing world: continuous improvements [227], fast TTM [228] and lifelong learning [229]. This applies specifically to software engineering that is in a continuous evolution because of innovation [230]. Employees are required to adapt to an ever-changing world [231].

We will now introduce continuous software development, lifelong learning and the relation between these two.

[1]This work was originally published as:

T. Theunissen, S. Overbeek, and S. Hoppenbrouwers, "Continuous Learning with the Sandwich of Happiness and Result Planning," in *26th European Conference on Pattern Languages of Programs*, New York, NY, USA, 2021. doi: 10.1145/3489449.3489974.

7.1.1 Continuous Software Development

The primary context for the presented patterns concerns continuously changing software, including technologies, processes, development and maintenance as can be found in industry. The relation between lifelong learning and continuous software development is that continuous software development often includes several iterations of the software product. For example, some parts, such as data, business logic, and the user interface may persist and other parts, such as a database, the programming language, and a software framework such as jQuery may be replaced [30]. We refer to this continuously changing state as CSD. The main characteristics of CSD [S31] are that:

1. it covers values, principles, practices, tools, procedures and processes from Lean [2], Agile [1], and DevOps [3];
2. it embraces activities from the whole life cycle of a software product, i.e. from concept to end-of-life. In addition to Agile and Lean software development, it includes maintenance activities. In addition to DevOps, it includes continuous architecting activities. Continuous Integration and Continuous Deployment (CI/CD) are part of it;
3. it considers the continuously changing state of the software product and progress, such as progressive insights (e.g., regarding process, design, implementation), changes in contextual factors, new features or requirements, bug fixes, or other unforeseen factors;
4. it distributes information about software development across multiple tools. There is no central repository for all relevant information. Because of high demands for fast time-to-market, process automation comes with a wide range of tooling for designing, developing, testing, deployment and monitoring.

Second, lifelong learning applies to industry practitioners who are confronted with new concepts, frameworks, technologies and communities. Furthermore, lifelong learning includes, besides professional development, also personal, relational and organizational development.

7.1.2 Lifelong Learning

Learning refers to knowledge, skills, and attitude [232]. Of these three, knowledge is relatively easy to assess by means of tests. Someone taking a test can get the maximum score by answering all answers correctly. Skills are less easy to test unambiguously, and scores typically fall in broad categories like completed or failed, sufficient or success. The hardest and and most ambiguously to assess is attitude, because it is based on culture, beliefs, and values that cause mentors and professionals to judge differently. The SANDWICH OF HAPPINESS refers to all three aspects of learning. Students and professionals are primarily stimulated to reflect on knowledge, skills, and attitude. Furthermore, any comfort or annoyance that influences the results or process can be reflected upon. Individuals can become better professionals by taking into account personal habits. As we live in an ever-changing world, and especially for engineers that live by creating innovative artifacts, both students and professionals have to keep up with these evolving creations [233]. This is reflected in CSD which implies continuous changes during the lifetime of software as a product. Unskilled people in particular tend to overestimate their own competences and underestimate problems. This is the so-called Dunning-Kruger effect [234].

A second aspect of lifelong learning is the continuous balance between skills and challenges, an increasing number of skills and challenges are in a balanced flow [235]. This is depicted in Figure 7.1. This flow starts from the first learning experience, includes learning years at school and at the university and continues for practitioners in the industry. The balance refers to related emotions staying between boredom and anxiety.

7.1.3 Patterns

The objective of this paper is to provide practical guidance for students and practitioners for continuous learning in CSD. It aims to be operationalized by a handout with practical usage and a theoretical background. To support this objective, in this section a brief conceptualization is presented for better understanding of the approach and patterns. In general, **a pattern is defined by a proven solution for a problem in a recurrent context**.

7.1.3.1 Alexandrian patterns

The 'pattern' concept was coined by Christopher Alexander in the context of problems when designing towns, buildings or windows. For Alexander, a pattern is defined by the following characteristics [236].

1. A **picture** which shows an archetypal example of the pattern.

2. An introduction that sets the **context** of the problem.
3. The **problem** in one or two sentences.
4. The **body of the problem**, including empirical background, evidence, and range of manifestations.
5. The **solution** is always stated in the form of an instruction.
6. A **diagram** of the solution that indicates the main components.
7. A **relation** of the pattern to other patterns in a pattern language.

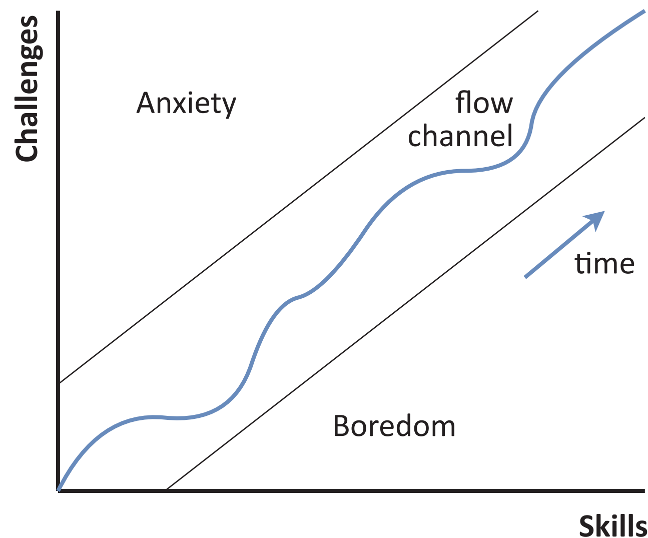


Figure 7.1: Flow is a channel between Boredom and Anxiety based on a balance between Skills and Challenges [235].

7.1.3.2 Coplien Patterns

Coplien [237] introduced this pattern [237] and was a co-founding member of the Hillside pattern community.

1. The **name** of the pattern.
2. The **alias** refers to alternate names or 'also known as'.
3. The **problem** describes the goals and objectives to achieve.
4. The **context** describes the preconditions or applicability.
5. The **forces** section lists the constraints, and interactions and trade-offs between the constraints.
6. The **solution** section instructs how to construct the product.
7. The **example** section illustrates a specific, easy to use, application of the pattern.
8. The **resulting context** section describes the consequences, post-conditions and side-effects of the pattern.
9. The **rationale** section justifies and explains the steps and rules how the trade-offs are applied.
10. The **known uses** section demonstrates the accuracy and firmness of the pattern for recurring problems.
11. The **related patterns** section relates to other patterns with common forces, initial or resulting context.

In this paper, we select the characteristics that are related to taking decisions: context, problem, forces, solution and consequences.

7.2 Our Two Patterns

The patterns Sandwich of Happiness (Section 7) and Result Planning (Section 7) have been used for several years at HAN UAS to support students working on school projects and in internships.

7.2.1 Pattern: The Sandwich of Happiness (SoH)

Continuous learning by answering three questions: what was good, what was bad, what could be better?

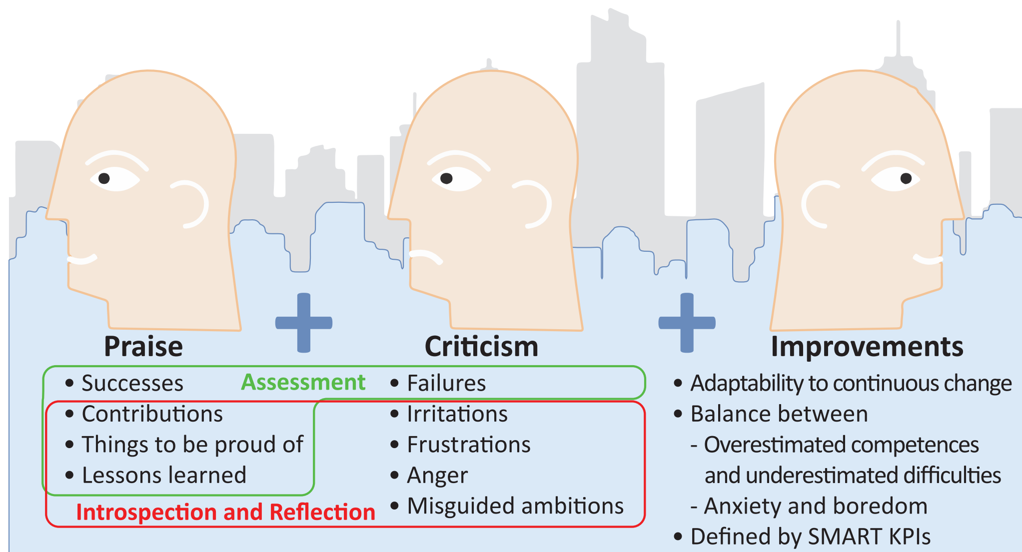


Figure 7.2: With the SANDWICH OF HAPPINESS, assessors look backward for praise and constructive criticism. Assessors look forward to improvements. The bitter pill is in the middle, sandwiched between a positive evaluation and a positive outlook. The assessment can be performed by a coach or as a self-assessment.

7.2.1.1 Context

An SoH is prepared several times during a process (e.g., in an iteration, at mid-term, during a semester or regular coaching sessions) because an evaluation only at the end of the project, without the possibility of improving during a project, would be rather pointless. Typically, it is hard for students and professionals to be critical of someone's performance as it may easily turn into non-constructive social dynamics that are hard to escape. The SANDWICH OF HAPPINESS aims at eliciting constructive criticism by looking back for positive and negative results as well as looking forward for improvement.

7.2.1.2 Problem

Learning, improving, and adapting to change is problematic because it involves cognitive aspects, performance, and attitude [232]. It is a continuous process [229] that starts at university, goes on in professional life and extends to retirement, in short: it takes a lifetime.

Developments in engineering are a continuous process of improvements, evolutions, and revolutions. This relates primarily to technology but extends to processes and societies where developments are used. A **mind shift** is required to keep up with or stay ahead of these developments.
Scrum, a widely used software development process, has **no mechanism for emotional hygiene** where developers can solve personal issues or build on personal characteristics.
There are demands from the market to stay ahead of **competition**. In an ever faster moving world with competition from every website, high demands for fast time to market require continuous adapting to change.
There are **Legal requirements** (e.g. GDPR). Demands for change might come from legislation to implement the requirements that might be unforeseen when designing a solution.
The team needs to **repair design flaws and bug fixes** to keep the product functioning as intended.
Feature requests from end users or customers require developers to continuously learn to understand end users and customers, to translate requirements into executable and manageable tasks.
There are **Progressive insights**. Once a whiteboard sketch is drawn in a team or a specification is written down, almost immediately improvements to the sketch or specification emerge.
Refactoring takes place: fixing code that is not broken but improving performance, readability or maintenance requires changes to the software product.
Peaceful and quiet delivery of requirements in a waterfall process is in strong contrast with high demands for continuous change.

7.2.1.3 Forces

Following Kruger and Dunning [234], **incompetence** makes it hard to reflect on inadequate performances. More in general, people overestimate their competences and underestimate the difficulties at hand.
Csikszentmihalyi, Abuhamdeh, and Nakamura [235] define a **balance between anxiety and boredom** that flows in time. See Figure 7.1.
It is more easy, and perhaps more kind, to give positive feedback, such as in tips and tops, than it is to **express irritations and frustrations**. Irritations, frustrations and misguided ambitions are not candidate for improvements and should be avoided.
Keep up with **realistic ambitions**. With all the failures, missed deadlines, overestimated competences and underestimated assignments, it is sometimes hard to be **proud of the positive achievements** such as contributions and lessons learned.

7.2.1.4 Solution

The resolving principles that are taken into account are depicted in Figure 7.2. The Sandwich of Happiness can be instructed as "three questions with three answers". Two questions look backwards, the third looks forward. For all three answers, use introspection and reflection to assess the results and the course for the results. With introspection, the person contemplates on his or her own actions and motives whereas reflection concerns other persons and context. The most specific objects for introspection and reflection are results, but tasks, processes and emotions can also be included.

1. *Looking backward*, start with writing down **positive results** such as your achievements and successes. Furthermore, reflect on other positive results such as contributions, and outcomes to be proud of. Even lessons learned in case of failures are positive.
2. Also *looking backward*, write down your **negative results** such as failures, and reflect on the causes, course and realization of how the objectives were not met. Irritations emerge in situations that do not contribute to productivity or happiness. Frustration relate to situations that someone can not change to improve. Ambitions that do not match with capabilities should therefore be avoided. Note that it takes some practice to be critical of ones own performance and also that it takes some exercise to be professionally 'cool' in dealing with negative results.
3. *Looking forward*: based on your results, the causes of the results, and your ambitions, define result goals for the next iteration. Typically, looking forward includes commitment to **positive results**. Looking forward includes contemplation and reflection on adapting to change, and keeping a balance between overestimated competences and underestimated difficulties, thereby finding a flow between anxiety and boredom.

7.2.1.5 Consequences

The objective for the Sandwich of Happinesss is a continuous improvement in an ever changing environment, such as in CSD. The time it takes to write down a Sandwich of Happiness is about 15-30 minutes. The text is concise, to the point, relevant, easy to read and makes it possible for a coach or mentor to do quick assessments on the results and reflections.

Because the Sandwich of Happiness is liberal in trade-offs between the forces, students can reflect on anything that influences the results. This includes who you are, what you want, your capabilities, barriers and limitations.

Furthermore, improvement does not imply an ever continuous upward path. It also includes -like a bull and bear market- downfalls, dips and drops. However, these downfalls as well as peaks lead to flowering advancements and evolution.

7.2.1.6 Examples

In Appendix 7, an example is presented of the template for the Sandwich of Happiness. The description that goes with the template is presented below.

The size of a typical Sandwich of Happiness is more than 1/2 A4 page but does not exceed the size of a full A4.

The Sandwich of Happiness is partly –under the first two questions– a thoughtful reflection concerning last week's accountability of the planning and results. The third question is looking forward to where you define improvements.

1. Looking backward, what was successful, what are you proud of, what have you contributed for others and to results, what was an example to others, what should be continued, what was positive?
2. Looking backward, what went wrong, what were irritations and frustrations, what was a misguided ambition, what must not be tried (happen) again, what should be stopped; in short: what was negative?
3. Based on 1) en 2), what will you do next week, what will you start with? Make it specific, for example SMART.

7.2.1.7 Related Patterns

There is a Feedback Pattern pattern described by Bergin, Eckstein, Volter, *et al.* [238], published in a collection of education patterns in 2012. The pattern describes two types of positive feedback and improvements in the middle for students to remain confident in their understanding. We consider this pattern to lack structure; it could be better thought through. Furthermore, we consider that while he pattern described by Bergin, Eckstein, Volter, *et al.* [238] focuses on students only, the pattern is also applicable to practitioners in industry in context of lifelong learning. Finally, the Feedback Pattern is part of Feedback Patterns in general. The feedback does not make a distinction between introspection (only take your own actions and motivations into account) or reflection (also take the context and others into account).

7.2.2 Pattern: Result Planning

Continuous improvement by being strict on planning and reporting of verifiable results and relaxed on accountability.

7.2.2.1 Context

This pattern is applied when working in processes that have the characteristics of CSD. It does not depend on the evaluation loop, feedback, phases, sprints, or planning/feedback loops. One of the values from agile is that working software is more valued than comprehensive documentation. Beck [1] assumes that all time is spent on developing software. This value ignores other tasks such as, for instance, –for both students and professionals– planning, meetings, administration, presentations, and –for professionals– acquisition and traveling.

It is hard to estimate the time spent on tasks other than development, for example by students who cannot lean from experience, or by professionals who have to deal with managers pushing fast time-to-market features. Students and professionals have tasks that do not directly contribute to productivity, such as all kinds of meetings or administration. Professionals might have additional tasks such as acquisition, training, and traveling. On top of that, switching context between tasks takes time to accommodate. When accountability for productivity and time is missing because of "unproductive" tasks, the time spent might become marked as unprofitable.

7.2.2.2 Problem

The problem with measuring results is that:
1. It is hard to define **verifiable objectives**. For instance in scrum, user stories and tasks can be defined. However, the formulation of definitions of done for user stories or acceptance criteria for tasks is often sloppy, if they are defined at all.
2. Results are typically expressed as **efforts**. Examples of efforts are activities such as 'reading', 'meeting', 'designing', and 'programming'.
3. No rules for **timing**. In scrum, there are no explicit rules how long tasks or iterations should take.
4. People may fear **failing** and taking **blame** when results are not accomplished. Usually, working software is not questioned but failing software can have valuable learning results.

7.2.2.3 Forces

The following forces are taken into account:
Method bonanza. There are many approaches in CSD that define how to plan tasks or results, such as Lean, Agile and DevOps. Students are taught textbook definitions but professionals often use best practices that leave out parts or introduce processes that fit for their purpose.
Death by planning. Popular tools such as Jira support the planning of user stories and tasks. However, extensive upfront contemplation of definitions of done and acceptance criteria keep students and professionals

away from committing to results.
Efforts instead of results. Results can range from a strict focus on SMART key performance indicators (KPI) on the one hand to just mentioning efforts on the other. It is both hard to *define* SMART results as well as to *achieve* results when defining efforts.
Programming is fun. Developers love the act of coding and tend to avoid diverting activities such as planning, documenting or process-oriented meetings.
Deviating from textbook definitions of methods. Students are taught textbook definitions of processes such as scrum, including all ceremonies. Practitioners from the industry hardly follow the paradigm from these textbook definitions but only use what is applicable in their professional context. It can be hard to grasp which aspects of methods are strictly necessary and which are optional.
Time estimation can be really difficult. Jones [239] shows that there is no scientific literature on estimating time for tasks. Furthermore, the combination of experience, (lack of) control of process and context, mitigation of risks and other potential threats, and creative characteristics of inventing solutions make it hard to give a reliable estimation for time.
When being held **accountable**, students and practitioners tend to find explanations that point to others or to external causes for failing to achieve results. The aim, however, is not to deal blame but to improve, and a full assessment can best accomplish this for the student or professional. For instance, one cannot blame another for being late delivering a requirement, but one *should* address the other for being late.

7.2.2.4 Solution

The resolving principles to balance the forces are related to delivering verifiable and transparent results. The forces for the Result Planning tend to involve individual accountability rather than team accountability.

The template for the Result Planning, as shown in Appendix 7 has columns for team member, results, achievements and, optionally, notes. The result statement refers to being concise and informative in single line sentences. Only the main result should be mentioned, so when there are more results for half a day, then the most significant result should be mentioned. Again, efforts are not allowed since these are always 'successful', i.e./ the effort taking place means success. For example, sometimes, students (as well as professionals) have things to do other than work, such as medical or driving lessons. However, it is mandatory in such cases that they find alternative time slot to do the work, even if this is in the evening or weekends. The achievement column is small and, when a result is achieved, contains a commit hash or link to some live document. In case of failure, there is only space for the core message of failure. Long descriptions are not allowed, to hide incompetence.

The time horizon is one week, and often students cannot oversee more than three days with tasks stretching in time. The template is used within a team and results are planned together.

The next week, each team member has taken two actions: 1) fill in the achievement column and 2) define new results. The result statements and achievements are, together with the Sandwich of Happiness, discussed with the teacher, coach or team lead.

7.2.2.5 Consequences

SMART(ish) results. Results are defined SMART(ish) so they can be measured. Activities like 'reading', 'programming', 'learning', 'attending (meeting/class)' and so on are by definition successful and stretch into the available time. Results like 'lists of concepts or frameworks', 'working software', 'asked questions, got the answers', 'updated Result Planning' and so on are examples of activities that can be measured.
Measure velocity. It is easier to check whether you make progress when main results are defined for half a day.
Smaller iterations. It is easier to both define moonshot goals, as well as realistic objectives, than it is to split the task up in smaller tasks that take risk, dependencies, and resources into account. Also, there is no scientific literature that prescribes an optimal duration for an iteration.
More significant results. Over time, students learn to define a more significant, relevant and contributing result in half a day. The period of 4 hours remains the same, but results are increasing. (In the beginning result are often too ambitious and expectations can not be met.)
Achieving goals makes people happy. Happy customers, happy developers and a happy team are within reach. Suppose a team consist of 5 people who define 2 results per day in an iteration for 1 week. That sums up to 50 results per week! Even when not successful for all projected results, this is impressive and encouraging.
Reliable. When using this pattern, you will become more reliable for yourself, team, teacher and manager

because you will be able to make better predictions concerning your knowledge, expertise and how you can use your experience.
Continuous learning and improvements. In CSD, this helps in adapting to change due to continuously changing demands from stakeholders or fast time-to-market, or changing constraints, risks or contexts. Additionally, learning experiences are higher in small iterations than in large iterations.

7.2.2.6 Examples

An example of a Result planning is shown in Appendix 7. The description that goes with the template is shown below.

Usage Per team member, you define a result per half working day. For a team with five people, you have 50 results per week or 100 results if your iteration takes two weeks.

Objectives What result do you want to achieve at the end of the day? Mention results that can be demonstrated, verified, or tested. Do not describe mere activities, such as 'reading', 'meetings', 'designing', 'programming' etc. but verifiable results. Typical results are a list of concepts, decisions, design or working software.

Achieved results Make sure that you can demonstrate verifiable results and these results relate to the planning:

1. Make links to GitHub for comments and code.
2. Make links to, e.g., Dropbox, Google Drive for design, documents and books.
3. Make links to websites.
4. Reflect on achieved results in your individual Sandwich of Happiness.

What went wrong? It is OK if an objective is not achieved. Do not spend more time than planned. Ask your coach, instructor or an expert for support.
Mention the reason for failure. There are a lot of valid reasons why you could not achieve the objectives. Be fair and try to be honest to yourself and your team. You will learn the most if you are critical towards your results and process.

The objectives of this pattern are to:

1. Use your experiences to learn (not to deal blame).
2. Every week you will have better estimations and will better close the gap between planning and results.
3. Make increasingly reliable predictions involving your knowledge, expertise and experience.

7.3 Acknowledgments

We want to thank our shepherd Christopher Preschern for shearing our sheep.

Appendix 1: Template for Sandwich of Happiness

Sandwich of Happiness (theotheunissen.nl/happiness, goo.gl/YyTYj5)

Week of year		**Date** (on Monday)	

The Sandwich of Happiness is partly –in the first two questions– a thoughtful reflection of last week's accountability of the planning and results. The third question is looking forward to where you define improvements.

1) What was successful, were you proud of, have you contributed to others, was an example for others, was positive?
2) What went wrong, where irritations and frustrations, was a misguided ambition, must not be tried (happen) again; in short: what was negative?
3) Based on 1) en 2), what will you do next week? Make it SMART.

1.) <Your reflection on positive outcomes; **Replace YELLOW with your text**>
2.) <Your reflection on adverse outcomes, do *not* mention improvements>
3.) <Fill in below>

Specific	
Measurable	
Acceptable	
Relevant	
Time-bound	

These reflections and improvements take more than half a page and not more than a full page.

Author	___
Student number	___

Figure 7.3: Template for Results planning. See also http://theotheunissen.nl/happiness

Appendix 2: Template for Result Planning

Group			Template url on
Start date			https://goo.gl/ujD5AI
End date			http://theotheunissen.nl/results
	Planning before		Accountability after
	Who	Planned results	Achieved results
Monday morning	Alice	Set up infrastructure	link to document
	Bob	List of concepts and literature	link to document
	Charlie	Login form	link to git commit
	Dave	CSS framework selection	Not successful. Too many options. Smaller step to decide first about an easy framework or advanced framework.
Monday afternoon			
Monday evening			
Tuesday morning			
Tuesday afternoon			
Tuesday evening			
Wednesday morning			
Wednesday afternoon			
Wednesday evening			
Thursday morning			
Thursday afternoon			
Thursday evening			
Friday morning			
Friday afternoon			
Friday evening			
Saturday evening			
Saturday evening			
Saturday evening			
Sunday evening			
Sunday evening			
Sunday evening			

Objectives

What result do you want to achieve at the end of the part of the day? Mention results that can be shown, verified or tested. Do not describe mere activities (like reading, meetings, designing, programming &c)

Examples:

Questions as a result of reading books, visiting websites or viewing videos; interview transcripts; UI wireframes; functional or technical requirements; bug solutions; contributions to other teams or community; change list in code or requirements; infrastructure; documents; functionality.

Group results together, or split results that it fits in a part of a day.

Achieved results

Make sure that you can show visible results and that results are related to the planning.

- Make links to github for comments, code.
- Make links to dropbox, google drive for design, documents and books.
- Make links to websites.
- Reflection on achieved results in your individual Sandwich of Happiness.

What went wrong

It is OK if an objective is not achieved. Do not spend more time then planned. Ask your expert for support.

Mention what the reason was for failure.
There are lots of valid reasons why you could not achieve the objectives.
There can be good reasons for failure or not so good reasons. Be fair and try to be honest to yourself and your team. You will learn the most of you be critical on your results and process.

Reasons for failure can be valid.

The objectives of this method are to:

- Use your experience to learn (not to blame).
- Every week you will have better estimations and will close the gap between planning and result.
- Make reliable predictions about your knowledge, expertise and experience.

Figure 7.4: Template for Results Planning. See also http://theotheunissen.nl/results

Chapter 8

Approaches for Documentation in Continuous Software Development[1]

Abstract It is common practice for practitioners in industry as well as for ICT/CS students to keep writing – and reading – about software products to a bare minimum. However, refraining from documentation may result in severe issues concerning the vaporization of knowledge regarding decisions made during the phases of design, build, and maintenance. In this article, we distinguish between knowledge required upfront to start a project or iteration, knowledge required to complete a project or iteration, and knowledge required to operate and maintain software products. With 'knowledge', we refer to actionable information. We propose three approaches to keep up with modern development methods to prevent the risk of knowledge vaporization in software projects. These approaches are 'Just Enough Upfront' documentation, 'Executable Knowledge', and 'Automated Text Analytics' to help record, substantiate, manage and retrieve design decisions in the aforementioned phases. The main characteristic of 'Just Enough Upfront' documentation is that knowledge required upfront includes shaping thoughts/ideas, a codified interface description between (sub)systems, and a plan. For building the software and making maximum use of progressive insights, updating the specifications is sufficient. Knowledge required by others to use, operate and maintain the product includes a detailed design and accountability of results. 'Executable Knowledge' refers to any executable artifact except the source code. Primary artifacts include Test Driven Development methods and infrastructure-as-code, including continuous integration scripts. A third approach concerns 'Automated Text Analysis' using Text Mining and Deep Learning to retrieve design decisions.

Keywords Agile, Documentation, Executable Knowledge, Just Enough Upfront, Machine Learning, Natural Language Processing

8.1 Introduction

With the rise of ubiquitous Agile software development methods and the continuously changing demands and contexts involved, documentation for sharing knowledge in software projects becomes more critical. However, the attention span for documentation reading, in general, has decreased dramatically [23]. In previous research [20], [21], we observed that developers do not want to write, others do not want to read, but having no

---

[1]This work was originally published as:

T. Theunissen, S. Hoppenbrouwers, and S. Overbeek, "Approaches for Documentation in Continuous Software Development," *Complex Systems Informatics and Modeling Quarterly (CSIMQ)*, vol. 32, pp. 1–27, 2022, doi: 10.7250/csimq.2022-32.01.

documentation at all is not an option. Therefore, the question is when the specification of requirements can be considered 'just enough' before starting or completing an iteration or a project. In this paper, we will address three possible approaches that contribute to answering this question in the context of CSD. CSD is defined as covering the values, principles, practices, processes, and tools from Agile, Lean, and DevOps. CSD covers the whole life cycle of a software product, starting from concept to end-of-life of a software product. Furthermore, it includes changing architecture, design decisions, operations, and maintenance to keep up with a continuously changing context, and changing demands. Finally, in CSD, knowledge about software products is distributed across multiple tools for software design, development, testing, operation, and maintenance. In Section 8, we

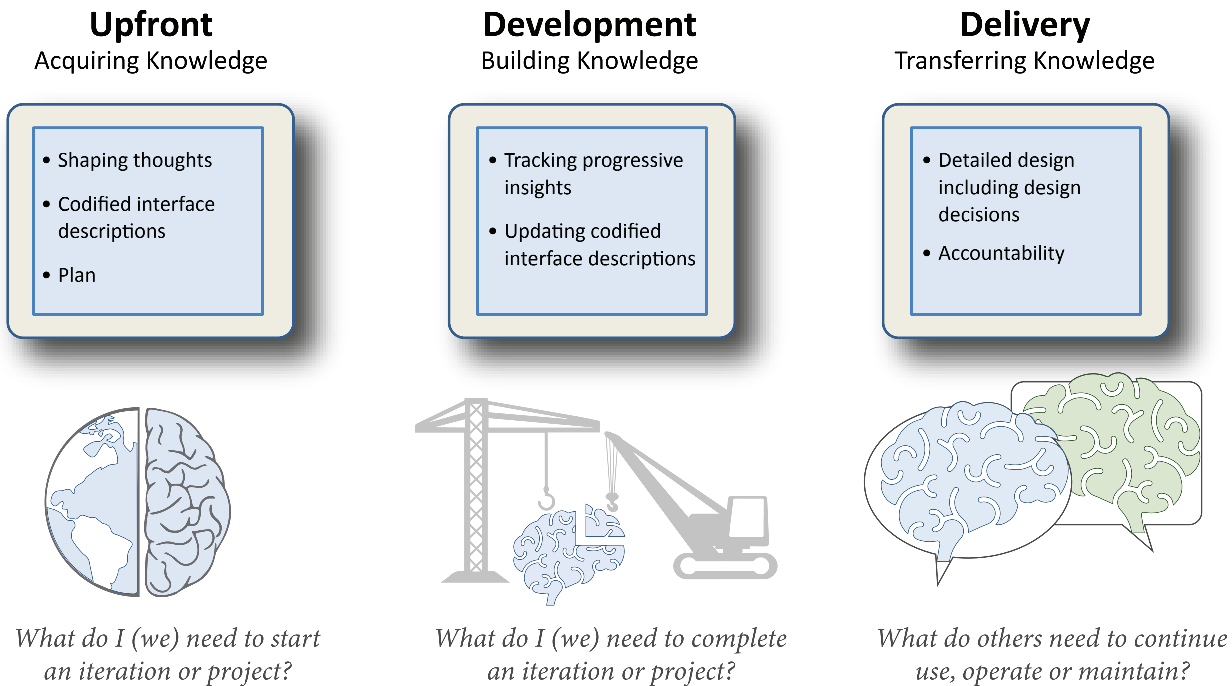


Figure 8.1: Phases with Knowledge Acquisition, Knowledge building, and Knowledge transfer.

will describe the conceptual research framework for constructing[2] the approaches. This framework applies to empirical sciences. We will define *empirical science* as any research where data is involved in distinguishing it from *theoretical sciences*. In theoretical sciences, where methods are the main focus, one strives to improve methods to obtain new knowledge or reasoning schemes.

There is a distinction between knowledge required up front to start a project or an iteration, knowledge required to deliver a project or an iteration, and knowledge required to continue a project. See Figure 8.1 for a diagram. When we refer to 'knowledge', we refer to all *types of information* as shown in Figure 8.2. The relation between information and knowledge, in this context, is that knowledge is a meaningful type of information for a stakeholder or system. In other words: information becomes knowledge if it contributes to comprehension, if it can be communicated (and discussed in case of humans) to other stakeholders or systems. Communication of information may vary from verbal conversations and whiteboard sketches to data and source code. The type of information that is required is related to the type of stakeholder and tools. For instance, developers require other information than operators or end-users. Not all types of information are required upfront before starting a project or an iteration. Furthermore, different types of information are created and retrieved by different tools, such as Git comments for natural language or whiteboards for sketches.

8.1.1 Previous Research

This study follows up on previous research, where we have found three candidate approaches. Figure 8.3 shows the studies in this research project and results from previous studies. The approaches in this study are the

[2]Because 'design' has many meanings, we use the term 'construction' for introducing the approaches. 'Design' in this study may refer to 'software design', 'design decisions', 'design phase', 'design science', or 'research design'. To avoid confusion, we use 'design' in combination with a contextual term.

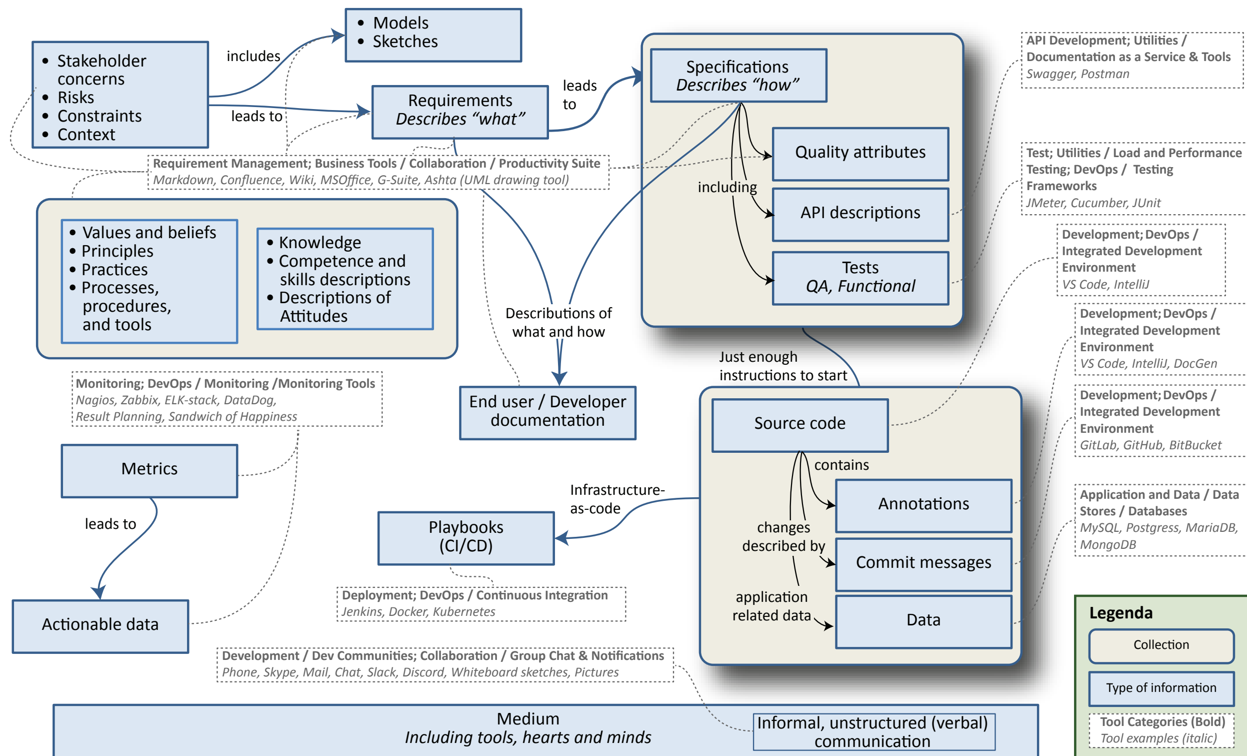


Figure 8.2: Types of Information including mapping to tool categories and tools [21].

elaborated recommendations from a previous systematic mapping study [20]. The recommendations from the previous mapping study are the practice of minimal documentation upfront combined with detailed design for knowledge transfer afterwards. In this study, it is named 'Just Enough Upfront'. The second recommendation from the previous study concerns executable documentation. The name in this study is the same. The third approach from the mapping study refers to modern tools and technologies to retrieve information and transform it into documentation. In this study, we focused on 'Automated Text Analytics' to retrieve design decisions from Git comments. A verification of the mapping study was conducted in a case study [21]. Furthermore, approaches are structured, the conditions, and characteristics, and artifacts are elaborated, and explicated.

We thus propose three approaches to keep up with modern development methods to prevent the risk of knowledge vaporization: 'Just Enough Upfront' documentation, 'Executable Knowledge', and using 'Automated Text Analytics' to retrieve design decisions. In this study, we will construct these approaches and elaborate on the requirements, characteristics, and artifacts that define them.

8.2 Contributions

The contributions of this paper apply to researchers in academia, professionals in industry and students and lecturers in learning communities. For all communities, the main contribution is the distinction that is made between what a developer needs upfront to start and what is required afterward to deploy, use, and maintain a software product. Along with this distinction come artifacts that are useful for the designated phases. Researchers in academia have investigated the artifacts, e.g., [S145], but have not made a clear and sharp distinction between artifacts that are of typical in use upfront, during, or afterward. Furthermore, using NLP with Automated Text Analytics to reveal design decisions is a relatively new research area. Practitioners in the industry have a way of working that deviates from textbook definitions, e.g. (large scale) Scrum, Lean, RUP, because of efficiency and pragmatic reasons that work well for them. However, this way of working is not validated or supported by management, and is not taught during education. Moreover, conceptualizing and optimizing the practical approach might increase productivity without suffering from knowledge vaporization. The third community this research contributes to is the learning community. Students in ICT and CS are taught to use big upfront designs -which makes sense for learning and experimenting with these methods- but are not taught the reasons why (or how) to deviate from textbook definitions.

In the remainder of this paper, the following subjects are addressed. In Section 8, the research questions and

research design are described. The approaches are introduced in Section 8. Section 8 describes the 'Just enough Documentation' approach. Next, in Section 8, 'Executable Documentation' is explained, and in Section 8 the approach with 'Automated Text Analysis' is described. In Section 8, the Threats to Validity are described. Finally, conclusions and future research are described in Section 8.

8.2 Research Design

8.2.1 Research Questions

The research questions are defined as documentation-related questions, which incorporate knowledge questions. The approaches follow a previous systematic mapping study [20] and a case study [21], and fit within the research cycle of a literature review [21], field research [20], the construction of approaches (this paper), and finally a validation of the approaches (future research). In Figure 8.3, the phases are depicted.

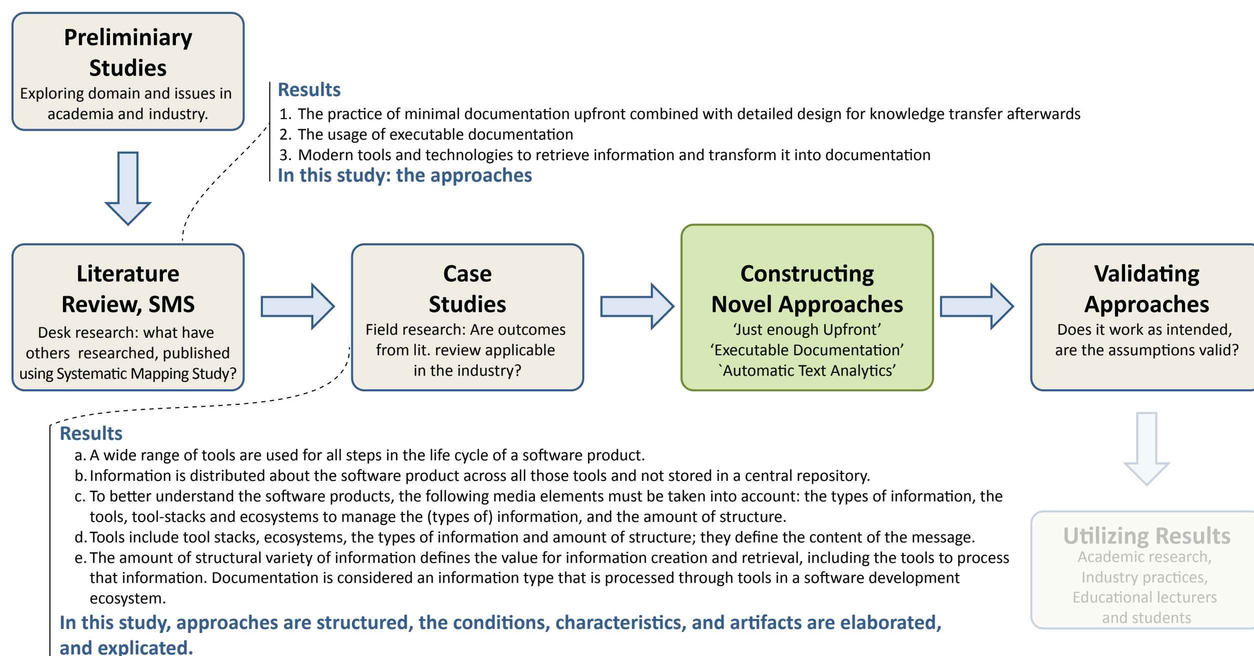


Figure 8.3: Studies in this research project.

The objective of this study is defined in the main research question: *What are the necessary and sufficient conditions to acquire, build and transfer knowledge about software products in CSD while threatened by increasing knowledge vaporization?* 'Necessary conditions' refers to the minimal requirements for an event to occur. 'Sufficient conditions' make the event to actually occur. A necessary condition alone is not sufficient. A simple example can make this clear. The necessary condition for delivering working software is a combination of 'knowledge', 'skills', 'attitude', and 'effort'. However, these necessary conditions for working software become sufficient when knowledge, skills, attitude, and effort are in a specific balance. A simple example for a sufficient condition without being necessary is "you used Angular instead of React for the front-end" where a framework is required but not which framework is actually used. 'Acquiring knowledge' refers to the knowledge that is required before starting a project or iteration. 'Building' refers to the progressive insights while developing the software. 'Transferring knowledge' refers to the knowledge that is required by others such as operators, maintainers, end-users, or new developers. With 'software product', we refer to all phases from concept to retirement; activities including design, architecting, development; and artifacts -both executable and non-executable. 'CSD' is an umbrella term for Agile, Lean, and DevOps values, principles, practices, and tools and processes. The term 'knowledge vaporization' refers to the practice of loose, natural, and informal communication about the software product.

The main research question leads to the following three research questions:

RQ1: For approach 'Just Enough Upfront' documentation to start a project or an iteration:

1.A. What are necessary and sufficient conditions to take into account for this approach?

1.B. What are the defining characteristics that distinguish it from other approaches?
1.C. What are the artifacts that are in use with this approach?

RQ2: For approach 'Executable Documentation' to transfer knowledge about a software product:
2.A. What are necessary and sufficient conditions to take into account for this approach?
2.B. What are the defining characteristics that distinguish it from other approaches?
2.C. What are the artifacts that are in use with this approach?

RQ3: For approach 'Automated Text Analytics' using NLP for retrieving design decisions:
3.A. What are necessary and sufficient conditions to take into account for this approach?
3.B. What are the defining characteristics that distinguish it from other approaches?
3.C. What are the artifacts that are in use with this approach?

8.2.2 A Conceptual Research Framework for Constructing Approaches

In order to account for our way of constructing the approaches, we first consider the distinction between theoretical sciences and empirical sciences. The main concern for theoretical sciences is striving for methodological improvements that enable the growth of knowledge and reasoning. The main concern for empirical sciences is delivering designs that start with discovering a problem and end with the invention of a solution. The framework that we use to construct the approaches involves the empirical science viewpoint, in particular that of software engineering. It is presented in Figure 8.4. The dynamic part depicts a flow that starts from discovering

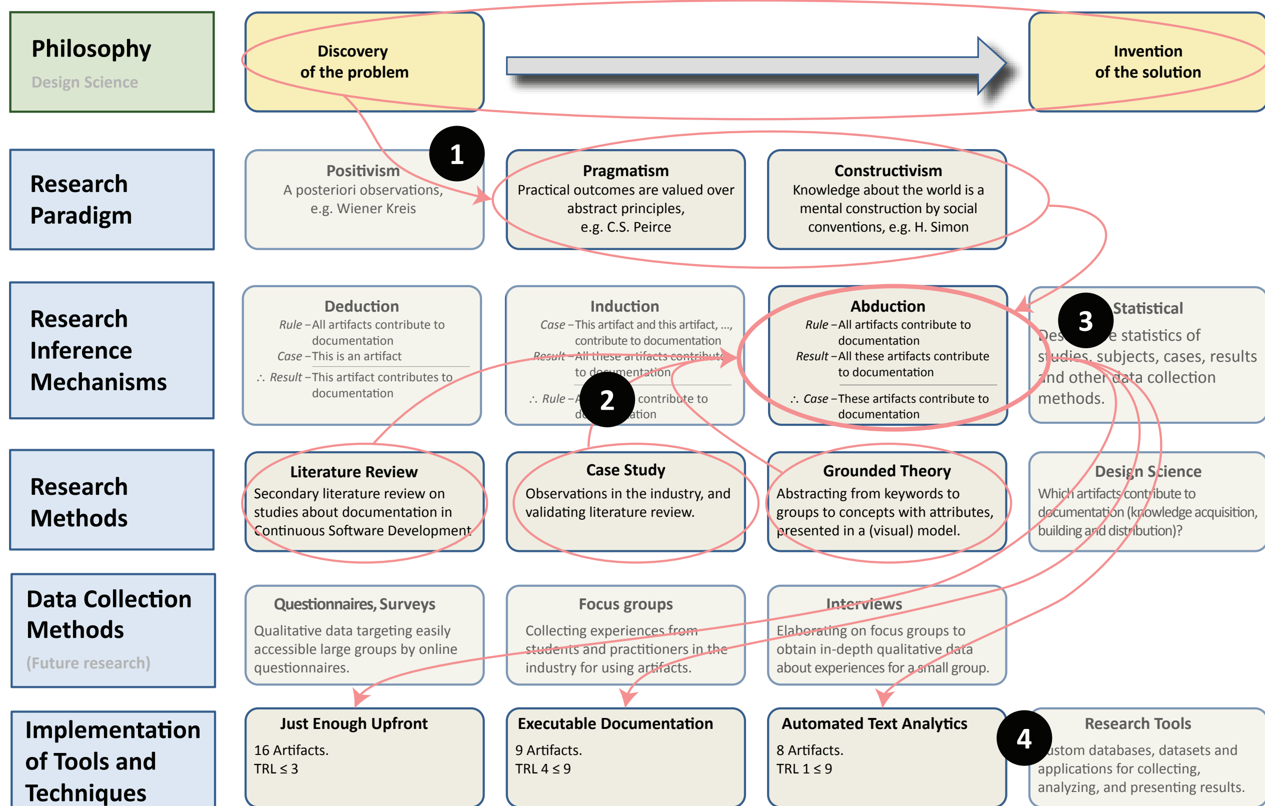


Figure 8.4: The Conceptual Research Framework.

phenomena that define the problem and ends with the invention of a solution. The discovery of phenomena refers to observations in a platonic (ἑιδος, eidos) universe, Kantian noumenal world but also observations of phenomena in the tradition of the British empiricists such as Locke and Hume. With the invention, actual changes are made in the context.[3] Obviously, these phases can pass several iterations. The research paradigms refer to a distinct set of structured beliefs and behaviors to address ontological and epistemological questions [240]. Researchers try to establish logical and causal relations between phenomena with inference mechanisms. With

[3]Compare this with the *discovery* of mathematical objects like numbers, cubes and spheres, and *invention* of complex numbers.

deduction, an explanation can be given for phenomena. This does not lead to new knowledge *in general* but only for the researchers involved. Induction might lead to new knowledge, but this inference mechanism is not logically valid if not all cases can be tested. With abduction, hypotheses can be falsified [241]. This is the weakest form of logical reasoning but is often used. Statistical inference mechanisms are mathematical approaches to describe events and are often used in empirical sciences such as engineering, social or medical sciences. Methodologies are distinct from methods in that the '-logy' suffix refers to an understanding and description of an *applied* method (that is, without the '-logy'). The research methodologies describe concepts about the collection and interpretation of observations. A systematic mapping study based on Petersen, Feldt, Mujtaba, *et al.* [103] and Kitchenham and Charters [222] has been conducted to investigate what others already have researched. In the case studies, we investigated the data in a practitioner's context (education and industrial) [215]. With grounded theory Stevens, Rohde, Korn, *et al.* [242] in the mapping study, we categorized data into groups, groups into categories, categories into concepts including relations between concepts. For this study, we use Design Science (DS) based on Wieringa [11], March and Smith [243], Simon [244], and Hevner, March, Park, *et al.* [245], we applied design science as a solution in practice. The techniques, tools, processes, and procedures in the approaches will be collected, analyzed, and presented to readers using diagrams.

The framework is used to discover the issues and invent the approaches. The conceptual research framework is not a linear step-by-step process but an exploration with successes, failures and iterations.

8.3 Approaches for Documentation in CSD

Based on previous research, the approaches in this study are refined using the conceptual research framework, as presented in Section 8. Design Science is used for discovering the problem and for invention of the solution [243]. A typical research, according to Wieringa [11], cycle includes a problem investigation, treatment design, treatment validation and treatment implementation [11]. From the research paradigms, constructivism is used [244]. A typical inference method for generating a hypothesis, i.e., approaches, is abduction [241]. A systematic mapping study and validation in the industry with case studies leading to a grounded theory have been conducted in previous studies [20], [21]. It resulted in the practice of minimal documentation upfront combined with detailed design for knowledge transfer afterwards, the usage of executable documentation, and modern tools and technologies to retrieve information and transform it into documentation. Data collection methods, including questionnaires, surveys, focus groups, and case studies, are used in previous studies [21] by structuring approaches, defining the conditions and characteristics, and elaborating on artifacts. For validation, inference to the best explanation, which gives the best hypothesis or theory for the given data, is used. A set of tools is created with software for data storage, analysis, and the presentation of results.

The first approach is 'Just Enough Upfront' to start and complete software design after completion of an iteration or project. From previous research [20], [21], [S31], it became clear that for upfront documentation, two necessary conditions must be met: shaping thoughts, and communicating interface descriptions between (sub)systems. For knowledge transfer, a representative software design is required. Most efficient is to create a fully detailed software design after all design decisions have been made and the software product is deployed and operational [3].

The second approach is 'Executable Documentation'. Traditionally written documentation is hard to keep up-to-date with the actual code (documentation generated from code or databases by reverse engineering is not considered because it is already documentation). Documentation cannot be tested and writing it is a tedious and intrusive task developers want to avoid. However, when writing requirements and specifications upfront that help verify, validate, and test the software product, the specifications can be human-readable —especially when using tools like Cucumber[4]. Furthermore, the human-readable specifications can be executed, so the document itself can be verified, validated, and tested. Because writing executable specifications shows great resemblance to the activity of coding software, it is not considered a tedious or intrusive task by developers.

The third approach concerns 'Automated Text Analytics' to retrieve distributed information about software products [21]. With this, we have two objectives in mind. The first objective is to extract relevant information from distributed tools for designated stakeholders. Such tools can range from PowerPoint to Git commit messages. The second objective is to understand the motives for decisions. Advanced technologies used include Text Mining and Deep Learning. The novelty refers to the distinction between knowledge –including values, principles, processes, procedures, methods, techniques, skills, and attitude– required *upfront* to start, and knowledge required *afterwards* to continue, as visualized in Figure 8.1.

[4]https://cucumber.io/

ID	Phase	Process	Artifact
D1	Upfront	Communication between stakeholders	• Yellow Pages.
D2 D3 D4		Shaping thoughts	• Presentation. • Whiteboard and drawings. • Lists.
D5		Communication between systems	• Interface Description Language.
D6		Planning	• Plan of Approach.
D7	Building	Progressive Insights	• Description of Concepts.
D8		Communication between systems	• References.
D9 D10		Coaching and Control	• Results Planning. • Sandwich of Happiness.
D11 D12 D13 D14	Afterwards	Deliverables	• Software. • Git Comments. • Full Detailed Design. • Decisions.
D15 D16		Accountability	• Compared Planning versus Actual Results. • Final Sandwich of Happiness.

Table 8.1: Phases, Processes, and Artifacts in Continuous Software Development.

The approaches are explored by applying the conceptual research framework (see Section 8 and Figure 8.4). The conceptual research framework for empirical sciences shows the process of designing the approaches, starting with the discovery of the problem and ending with the invention of the solution. The maturity levels of the approaches are adopted from the Technology Readiness Levels (TRL) [246]. (1) Refers to the two research paradigms: pragmatism because of abduction and constructivism because of design science. (2) Refers to the research methods from our previous research: systematic mapping study, case study and grounded research. With (3), abduction is used as inference mechanism. In (4) the approaches are presented.

8.3.1 'Just Enough Upfront' Documentation to Start a Software Product, a Project Or an Iteration

This approach results from a prior study [20]. The conditions, characteristics, and artifacts are elaborated, explicated, and structured for this approach. When using this approach, the bare minimum to start -and complete- an iteration is presented. Note that this is the opposite of traditional big upfront software design. Some domains are excluded from this approach because of regulations or legal requirements for documentation such as governments, food and drug administration, or the military.

This approach answers two questions. The first question concerns *current* team members and answers which knowledge is required to start a project or iteration. The second question concerns which knowledge is required for *new* team members and others who did not participate in the design of software products to continue development or operations. This includes end-users, maintainers, operators or new team members. In Table 8.1, the artifacts are presented that are in use with this approach, as derived from previous research [20]. They will be discussed below.

Conditions

— The approach for 'Just Enough Upfront' documentation to start a project or an iteration does not require specific definitions upfront.

Characteristics

— Typical of this approach is that it applies to exploratory projects with a TRL lower than or equal to three where an idea must be validated in a Proof of Concept (PoC) [246]. It fits within the Agile philosophy that working software is valued over comprehensive documentation [1]. It follows the Lean principle that anything that does not contribute to the end-product is considered waste [2].

Artifacts

— The enumerated artifacts in this section refer to the three phases of a software project where these artifacts are in use: knowledge required upfront, required while building, and required afterwards for continuation. Key for the artifacts are the acquisition, building and distribution of knowledge (especially design decisions) that

increase productivity and fit within the development processes. The following artifacts are defined for this approach.

D1. *Yellow Pages.* This presents an overview of documents, ordered by type, phase and process in CSD. See Figure 8.2 for an overview of types of documents. This presents an overview of documents, ordered by type, phase and process in CSD. See The audience for this overview comprises all stakeholders. There is no specific template. Typical tools are web pages as a starting point from, for example, Confluence and GitHub pages.

D2. *Presentation.* The presentation aims at a good mutual understanding between stakeholders about mission, vision, strategy, and objectives. It is not possible to communicate meaning between sender and receiver, only symbols, as Shannon and Weaver [5] already pointed out. Furthermore, it is also not possible for a sender to enforce behavior by a receiver, at least not in an ethical way. Storytelling is a way of communication that is not exact, engages the audience, and introduces the issues at stake, usually starting with the solution supported by evidence and reasoning. A starting point to structure the presentation is by the format of Situation, Complication, Question, Answer (SCQA).

D3. *Whiteboard sketches and drawings.* These include all whiteboard sketches, drawings and visuals that assist in understanding and communicating objectives, approaches etc. The sketches and drawings are part of the presentation. From previous studies, it became clear that a format such as UML is not actually required [20], [21]. Basically, any box-and-line diagram that conveys an idea and achieves mutual understanding will do [247]. Ainsworth [248] made clear that a visual diagram leads to better understanding of relations (causal, logical). Text representations are better for a deeper understanding.

D4. *Lists.* We have to investigate if lists are a valid approach to deal with complex decision making and establishing priorities within the dynamics of Agile teams [249], [250]. The following aspect characteristics are based on Agile decision making as discussed by Rouse [251].

1. *Priorities.* Refer to the primary objectives for this software product. This includes a description of what criteria determine order and how they refer to achieving the goals.
2. *Long list.* Describe selection criteria. Describe the relationship between objectives and selection criteria. Refer to sources for the long list, e.g., Google trends[5], benchmark sites such as databases[6], jobs[7], trending technologies on Gartner hype cycle[8], and Thoughtworks[9]. Other aspects such as economic, legal, social, environmental, etc., are valid as well [252]. The number of items on the long list varies between 15 to 25, depending on context.
3. *Shortlist.* Make clear what defines the scope of relevant techniques. The number of items on the long list varies between 3 and 5, depending on context. A shortlist with only one item and no alternatives is not convincing.
4. *Features of framework, library, tool, process, and the like.* Describe the distinctive features for each item, including the fit for purpose. These distinctive features as such are not positive or negative. These features become an advantage or disadvantage when there is also an explicit judgment on the contribution of features to the objective.
5. *Comparison.* The feature comparison is a table with features on one dimension and techniques on the other, showing an evaluation of suitability. Also, adding a -kind of- scoring such as yes/no, a scale of 0-5, or -/0/+ contributes to getting grip on the matter. Scoring is an indication and not a calculation, so it is not a spreadsheet exercise but can be useful for quantitative analysis and support qualitative analysis.

The audience include stakeholders and development team.

D5. *Codified Interface Description.* This is a codified, formal document that describes endpoints, types, paths, filters, and variables between modules, components and (sub)systems. It includes the response time, such as in real-time batch or queuing mechanisms. Architectural patterns include pub/sub messaging in event driven architectures, protocols such as Representational State Transfer (REST), SOAP, JavaScript Object Notification (JSON) or XML-RPC, and technologies such as Common Object Request Broker Architecture (CORBA), Apache Kafka or Message Queuing. The audience is the development team, or a team related to external systems. The best templates are tools to manage interfaces such as REST API descriptions. These tools can be found, for instance, on https://swagger.io/, https://www.apollographql.com or any other Interface Definition Language (IDL).

[5]https://trends.google.com
[6]http://db-engines.com/en/ranking
[7]https://www.indeed.com/
[8]https://www.gartner.com/en/research/methodologies/gartner-hype-cycle
[9]https://www.thoughtworks.com/radar

D6. *Plan of Approach.* Elaborating on the systematic review by Abrahamsson, Warsta, Siponen, *et al.* [253] and empirical research by Dybå and Dingsøyr [254], the main contents of a plan in CSD are:

1. Results. This includes delivery dates, quality criteria, and methods for securing results such as a definition of done (user story), acceptance criteria (tasks), or other SMART definitions.
2. Resources. This includes prerequisites such as tools, licenses, and access to experts.
3. Constraints. These refer to decisions from the past that affect current decisions.
4. Risks. We identify two categories of risk: lack of insight and a lack of control. A lack of insights refers to situations where you have to make decisions without sufficient information. A lack of control refers to situations where you can not carry out interventions. Risk mitigation strategies and contingency plans must accompany the description of risks.

A plan usually changes with progressive insights. For *accountability*, the original plan is required to compare planned results with actual outcomes. Contributors and users of a plan are developers, product owners, and managers. Tools such as Trello[10] or Jira[11] are preferred for documenting and tracking user stories, tasks, or issues.

D7. *Description of Concepts.* This refers to mental presentations such as 'beliefs,' 'doubts', or any other relevant notions used in understanding, reasoning, communication, and discussion in order to understand the subject matter better or convince others to carry out the desired behavior. Typical in CSD are values (as expressed by actions and behavior of a team or community), principles (the explication of values in writings such as mission/vision/strategy statements or a code of conduct), best practices (experiences from the past with the desired outcome), and tools, processes and procedures (means and guidelines to perform actions). Any template that assists in reaching these objectives will do. Commonly in use for developers in CSD are 4+1 from Kruchten [S178] and C4 from Brown [210]. The audience is the team members.

D8. *References.* Typical for Agile projects are progressive insights. To keep track of these, a description of new and redefined concepts is required. When it comes to new concepts, to get acquainted with the subject area and to find a balanced view on it, a literature review Kitchenham and Charters [222] and a systematic mapping study Petersen, Feldt, Mujtaba, *et al.* [103] are useful. Additionally, when much is published in blogs, reports or websites other than peer reviewed journals or conferences, then the guidelines for gray literature from Zhi, Garousi-Yusifoğlu, Sun, *et al.* [255] can be helpful.

Striving for completeness involves the following types of inferences:

1. Authentic references identify publications where a concept is introduced or coined and that are the oldest publication to do this.
2. Authoritative references are the ones with the highest number of citations. Note that inherently, older papers often have more citations than new publications.
3. Actual references are those that have been trending in the last few (two to four) years.

D9. *Result Planning.* The Result Planning (RP) is planning of verifiable objectives and a short description of achievements [33]. The achievement description is either a link to a repository, a link to a live document, or a short description of why the objective is not met. The primary goal for the RP is continuous improvement by being explicit on objectives and accountability. The audience are coaches and developers. A template is available at https://theotheunissen.nl/results.

D10. *Sandwich of Happiness (SoH).* This is an introspection, reflection, and outlook on the realization of results and processes [33]. This can range from a commitment to results, personal thoughts, social aspects, or any other factors that had an effect. The primary audience is coaches and developers. An extended description and template can be found on https://theotheunissen.nl/happiness

D11. *Software.* This refers to executable files including source code, scripts, and executable specifications such as CI/CD, DDL, or DML. Contributors are the development team, and users of the documentation are operators and new team members.

D12. *Git Comments.* These refer to a meaningful description in natural language of modifications in the source files. It is not required to explicitly describe the differences between old and new code because they can easily be found by comparing commits. The audience is the development team, including new team members. Templates can be found on https://github.com/devspace/awesome-github-templates

D13. *Full Detailed Design.* The full detailed software design is created *after* a significant iteration: only after completing the software product in an iteration, an accurate description can be made. Any time sooner might result in inaccurate descriptions because of potential changes, such as progressive insights.

[10] https://trello.com/
[11] https://www.atlassian.com/software/jira

The purpose of the software design is to distribute knowledge about the delivered state of the software product. This design includes decisions with alternatives. The audience is stakeholders, especially the development team, including new team members. Any template can be used that follows the traditional Software Requirements Specification (SRS) [256], Software Architecture Description (SAD) [90] and Software Design Description (SDD) [91]. Also, C4 [210] or 4+1 from Kruchten [S178] can serve as template.

D14. *Decisions.* With decisions, an assessment is made of, based on argumentation. 'Forces' influence the decisions and can be a trade-off. For example, user-friendliness and safety are sometimes contrary forces. Stakeholders make decisions, and the audience is the stakeholders in a category for which the force is relevant, e.g., customers, developers, end-users, managers. Architecture decisions are typically hard to make at the beginning of the design of a software product, but costly when changed later in the process. Templates can be found in the pattern community[12], Architecture Decision Record (ADR)s[13], or Decision Centric Architecture Review (DCAR) [257].

D15. *Comparing Planned versus Actual Results.* Accountability of the outcome follows when comparing the plan of approach with the achieved results, for budgetary, contractual, and performative reasons. Typically, the bare minimum is delivering what has been agreed upon. However, typical for CSD projects is the use of progressive insights that often lead to redefining objectives or approaches. Keep in mind that a Minimum Viable Product (MVP) is to be learned from, not to shipped as with a Minimum Marketable Product (MMP) [258]. Typical assessments are comparisons between assignments and delivered results. Next are planned versus actual resources such as time and Full-time Equivalent (FTE), budget, knowledge, skills—furthermore, the comparison of constraints that reveal implicit choices or tacit knowledge that has been used. The last assessment is the actual management of risks. The difference between this comparison and the RP (see D9) is that this comparison applies to the completed life cycle of a project or software product.

D16. *Final Sandwich of Happiness.* See D10 for a description. It differs from its intermediate counterpart in that its scope of time is larger than with the SoH. It now applies to a period covering multiple iterations. The length of the final SoH lies between a half and full A4 page. The positive and negative items and improvements apply to observations between iterations, or to the most significant actions. The audience are peers in the team and coaches.

This approach, 'Just enough Upfront', answers the first research question. It counts 16 artifacts, of which whiteboard drawings, a Codified Interface Description, a Plan of Approach, and Design decisions are a few of these. There are no specific conditions for this approach. It can be applied to any maturity level of a software product and from parts of a software product to a complete operational system. Typical characteristics of this approach are exploratory projects where concepts, requirements, or specifications are not well defined. In terms of TRL, this applies to a PoC, levels smaller than or equal to three. In other phases, such as prototype, pilot, and production, it fits for Agile processes where working software is valued over comprehensive documentation.

Furthermore, it answers questions about "what is required upfront for an individual developer to start?" and "what is required afterward to continue, deploy, maintain and use the software product?" Most significant is that big upfront design is instead a throw-away document than transferable knowledge on the software product.

8.3.2 'Executable Documentation'

With Executable Documentation (ED), we refer to any artifact related to a software product except the source code, that defines, transforms, or distributes knowledge that can be executed. In Table 8.2, artifacts related to ED are listed.

Furthermore, ED can be tested because it can be executed. Traditional documentation in Word or Wikis, such as Confluence, can be validated for syntax and grammar, but this is not related in any way to the executable source code. ED can be executed just as any other executable source code. Next, ED is non-intrusive. Developers like writing source code, not writing documentation. With ED, the activity of coding has the same characteristics as writing documentation.

As shown in Figure 8.1, knowledge acquiring, knowledge building, and distributing knowledge also applies to executable documentation.

[12] https://wiki.c2.com/?PatternCommunity
[13] https://adr.github.io/

ID	Phase	Process	Artifact
E1	Upfront	Defining *what* has to be done.	• Executable Requirements.
E2		Defining *how* it has to be done.	• Executable Specifications.
E3		Using knowledge from previous experiences.	•Templates, Frameworks, Libraries, APIs.
E4		Quality control.	•Tests (TDD, BDD, Acceptance Test Driven Development (ATDD)).
E5		Management.	• Definition of Done, Acceptance Criteria, Specific, Measurable, Acceptable, Relevant, Time-bound (SMART) KPIs.
E6	Afterwards	Speeding up the CI/CD cycle.	• Infrastructure-as-code.
E7		Retrieving knowledge about software products.	• Reverse Engineering.
E8		Saving executable knowledge for future development.	• Enhanced Templates, Frameworks, Libraries, APIs.
E9		Accountability	• Accountability and Actionable Data.

Table 8.2: Phases, Processes, and Artifacts in Continuous Software Development. All artifacts refer to executable code.

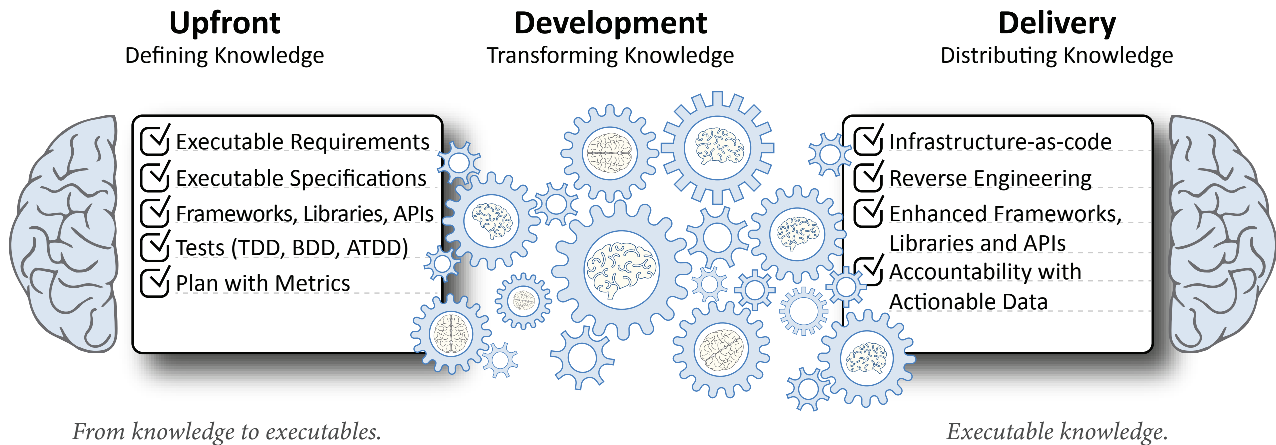


Figure 8.5: Executable documentation in consecutive phases.

Requirements

— Using Executable Documentation for development implies a well-defined set of requirements and specifications. It is not efficient while exploring an idea to start with tests because it would take too much time to re-iterate over the tests for what and how requirements and specifications would act.

Characteristics

— Defining characteristics of ED are that it is never out-of-sync, and it is just another representation of the software. Inline documentation within executable code is not part of ED as it is the same type of description as external documentation such as Word. Furthermore, documentation within source code is often not updated when the code itself is updated and also out-of-sync. Areas where ED is not the best option are situations where both problem and solution have to be explored, such as in TRL level 3 or lower for defining a PoC. Because of the many iterations in an exploratory phase, the approach with ED might take too much time, and the effort may therefore be too costly. Fast TTM is typical for projects that need to be ahead of the competition and keep pace with legislation [259].

Artifacts

— The following artifacts are defined in this approach.

E1. *Executable Requirements.* Requirements in a design define *what* a software product aims to achieve, such as business goals, legal obligations, or societal objectives. This applies to all phases of a life cycle of a software product, including understanding, simulating, implementing, deployment, operations, and retirement [260]. Requirements should be described in such a way that all stakeholders are able to

understand what is meant by them. Typical testing approaches are BDD, and ATDD where the input and output of a system is tested. Generally, requirements should be understood by all stakeholders.

E2. *Executable Specifications.* Specifications define *how* a software product is designed to achieve objectives and are an elaboration on the requirements. Specifications are typically defined for developers to understand and implement. A standard testing approach for specifications is TDD where the components of a black box of a system are tested.

E3. *Templates, Frameworks, Libraries, APIs.* These are examples of knowledge, best practices, or procedures that have a track record, saved as executable code, that gives development a head start. The template, framework, library, or API can be improved with progressive insights at the end of an iteration. A library is an arbitrary set of methods or procedures that developers can use and that typically does not prescribe a specific way of using it. Examples of libraries as a set of functions are graphical or mathematical libraries. A framework is a cohesive set of methods that operate together and are designed by a specific philosophy. Examples are front-end frameworks (Angular[14], React Native[15]) or Object Relational Mapping (ORM) frameworks such as Hibernate[16] or Sequelize[17]). The template has the most structure and least freedom for developers. All knowledge about the system is reduced to a set of questions a developer has to answer to install or operate a system. Typical examples for templates are questions asked during the installation of a software package.

E4. *Definition of Done, Acceptance Criteria, SMART KPIs.* A Definition of Done is used with user stories to verify that all requirements have been met [261]. A typical form of a user story is 'As <role>, I want <feature> because of <rational>'. A Definition of Done (DoD) is a rather formal definition of what the objective is . However, it is not specific enough to verify that criteria have been met on delivery of the fuzzy and not specific definition. In Scrum, a user story is usually split up in workable tasks. The validation of the criteria for tasks are defined in acceptance criteria. These are much more specific than in a DoD. Acceptance criteria can be defined in tests such as TDD or BDD.

E5. *Tests (TDD, BDD, ATDD).* An arbitrary categorization for tests is following the categories as mentioned in ISO/IEC-25010 [205]. This includes functional and non-functional categories such as scalability and security.

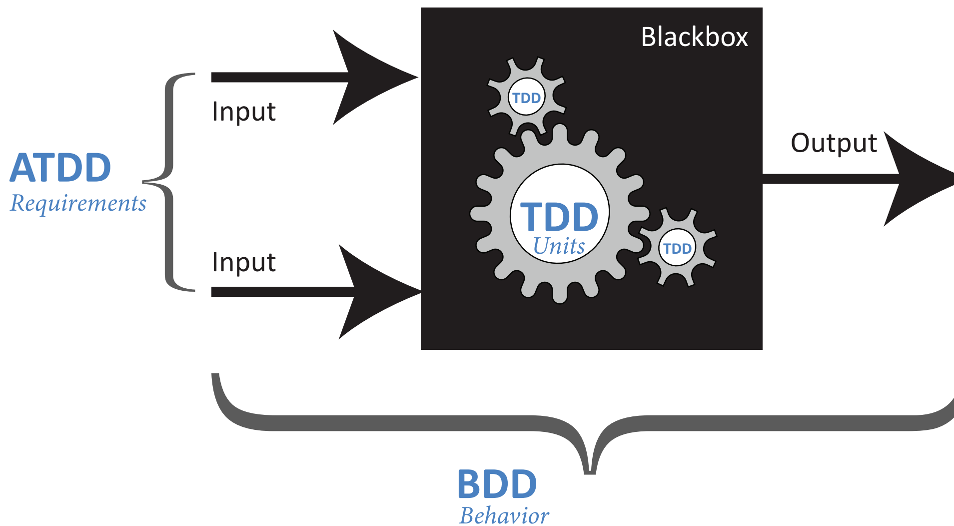


Figure 8.6: TDD, BDD, and ATDD and where they do apply to a software product.

TDD refers to unit tests to assist developers validating specifications. It focuses on small parts of the system that typically are part of the black box of a system [59], [262], [263].
BDD is is the testing of the behavior of the system, based on related input and expected output[18] [264], [265].
ATDD refers to capturing and validating requirements by analyzing user stories [266], [267], [268].

[14] http://angular.io/
[15] https://reactnative.dev/
[16] https://hibernate.org/
[17] https://sequelize.org/
[18] https://www.behaviourdriven.org/

E6. *Infrastructure-as-code.* This concept is about the automated testing, integration, deployment, and delivery of source code, commonly in use the CI/CD and DevOps community [73]. The code refers to scripts that are easy to read for humans or developers without the need to have explicit knowledge about the systems. A typical language is YAML[19], a relatively easy-to-read mark-up language. Examples for provisioning a system are Ansible[20] or GitOps [269].

E7. *Reverse Engineering.* With reverse engineering, we refer to any design artifacts that can be retrieved or constructed by analyzing, in retrospect, some source code, database DDL or DML, API, or infrastructure. Basically, anything that can be reverse-engineered is not required to be specified because it can always be produced, the source code being the 'single source of truth'. What is missing, however, are descriptions and decisions. Remember that documentation in source code is not reliable because this can easily be out-of-sync. Whenever executable code is updated, the in-line documentation is not necessarily updated.

E8. *Enhanced Templates, Frameworks, Libraries, APIs.* The enhancements refer to the new iteration of the template, framework, library, or API. Nah, Faja, and Cata [270] and Ghezzi [271] describe five categories of maintenance that apply to the evolution of software which applies to CSD too. These are:
   1. Corrective maintenance, indicating fixing bugs and correcting faults.
   2. Adaptive maintenance, implying new features and new demands in a changing context.
   3. Perfective maintenance, which refers to refactoring code, optimizing code, or improvements.
   4. Preventive maintenance, which refers to refactoring code to keep up with possible changes in the future.
   5. User support points at assisting users in using the system.

E9. *Accountability and Actionable Data.* Accountability is closely related to metrics on DoD for user stories, acceptance criteria for tasks or other kinds of (SMART) verifiable results. Furthermore, the deviation must be explained when planned results do not match achieved results, either positive or negative. Following Theunissen, Overbeek, and Hoppenbrouwers [33] concerning the Result Planning and Sandwich of Happiness, it does not make sense to have a plan of approach without taking accountability. It does not make sense to take accountability when only reporting events without planning.

   Actionable data refers to a subset of big data that, by (automated) analysis, is transformed into insights that require immediate action which can be acted upon [272]. Typically, it is both more effective and efficient to influence causes than being responsive to effects. However, when (external) causes are not under control, the effort is to mitigate unwanted results or elude a backup plan.

'Executable Documentation' answers the second research question. It counts nine artifacts, of which frameworks, templates, libraries, and APIs are a few of these, including TDD and BDD, and infrastructure-as-code. Conditions for this approach are that concept, requirements, and requirements must be well-defined upfront. It can be applied to any maturity level of a software product and from parts of a software product to a complete operational system. Typical characteristics of this approach are that it is used in pipelines for CI/CD, is fit for DevOps, and fast TTM. It does not apply to PoC where requirements and specifications are not well defined because it would take too long to develop requirements, specifications, tests, and code. The maturity for the phases are prototypes, pilots, and production.

This approach, 'Executable Documentation', makes clear that question about "what is required upfront for an individual developer to start?" and "what is required afterward to continue, deploy, maintain and use the software product?" assumes that requirements and specifications are well defined. Most significant is that big upfront design is instead a throw-away document than transferable knowledge on the software product. In terms of TRL, this applies to a prototype and pilot, levels larger than four.

8.3.3 Using 'Automated Text Analytics' for Retrieving Design Decisions in Different Types of Information

This section explores automated text analytics as a candidate approach for automatically capturing unstructured information such as natural language. The natural language entities we focus on are Git commit messages. The reason for focusing on Git commit messages is that this type of information is commonly available with source code while, for example, white-board sketches are not publicly available. Furthermore, the source code can be read for *what* it should do and *how* it should execute the code. The reasons for modifications, however, cannot be read from the code. The reason for the change is outside the code in tools such as Jira for tasks or Confluence for epics and explanations. Most close to the source code is Git as a source control management

[19] https://yaml.org/
[20] https://www.ansible.com

ID	Phase	Process	Artifact
A1	Upfront	Source Code	• Git Comments.
A2	Building	Text Mining	• Search Categories and Search Terms.
A3			• Statistics: Bag of Words (BoW), Term Frequency (TF), Inverse Document Frequency (IDF), Term Frequency-Inverse Document Frequency (TFIDF).
A4			• Annotated data.
A5		Deep Learning	• Model.
A6			• Pretrained Model or Transfer Learning.
A7			• Hyperparameter Settings.
A8	Afterwards		• Finding Design Decisions in different Types of Information.

Table 8.3: Candidate Artifacts in Automated Text Analytics.

system. The Pull Requests are most helpful in documenting *why* a change has been committed. It should be investigated if a reason for the change should accompany every commit.

In Table 8.3, the artifacts are shown that are in use with this approach.

After the 'AI Winters', occurring because of failure to deliver on promises [273], [274], ending around 1993, a range of neural networks came into existence such as Convolutional Neural Network (CNN)s, [275] for image processing and Recurrent Neural Network (RNN)s [276] for language processing, visually beautifully explained by Veen [277]. Figure 8.7 presents text mining and deep learning as a pipeline for language processing.

A special note for this section concerns the documentation of the artifacts itself. The artifacts do reveal knowledge about the software product, but need to be documented itself as well.

Requirements

— Using NLP for retrieving design decisions from Git comments requires verbose and structured comments. Typically, Git comments describe *what* the change is and not *why* the change is made [278]. What has been changed is easily retrieved by comparing the diffs or applying the rule that source code is the single source of truth (SSOT) [279].

Characteristics

— Using NLP for retrieving design decisions by automatic extraction of causal relations from natural language as Git comments is a relatively new area of expertise. In a secondary study, Yang, Han, and Poon [280] points to machine learning using advances in statistical text analytics that come available. We consider it applicable in all development processes, including waterfall, and not limited to iterative, incremental development processes. It also applies to all maturity levels of the TRL.

Artifacts

— The following artifacts are defined in this approach.

A1. *Git Comments.* Git comments have already been discussed in D12. For this approach, the Git comments are the source for the data analysis. The NLP term in use is 'corpus' (D). Candidates for the individual 'documents' (d) are the comments or branches. N refers to the number of documents in D.

A2. *Search Categories and Search Terms.* Search categories are closely related to types of modifications. Motta, Gomes e Souza, and Sant'Anna [281] summarizes four categories: Architectural Description Languages (keywords of the five most cited ADLs in Google Scholar), Not-Functional Requirements (the top seven emergent topics in Google Scholar), Architectural Styles (from the two most cited books in Google Scholar on software architecture), Architectural Constraints and Related terms. From these categories, a list of 452 search terms can be derived. In Table 8.4, commit types based on Anonymous [278] are shown.

A3. *Statistics: BoW, TF, IDF, TFIDF.* Table 8.5 presents an overview of statistical text mining methods in use. The search terms are processed by a stemmer and a lemmatizer, as well as all the text from the Git commit messages. A stemmer is a rather rigorous -compared to a lemmatizer- chopper that cuts common prefixes or suffixes from inflected words. For the English language, the Porter stemmer is in common use [282]. A lemmatizer takes into account the morphological analysis of words. One of the relevant distinctions between both methods for this approach of documentation is that stemming is less important for meaning whereas lemmatization takes meaning into account.

A4. *Annotated Data.* This refers to classification and labeling data to identify features for a knowledge domain. Based on this limited set of annotated data and other settings, the neural network can process unprepared data.

ID	Title	Description
build	Builds	Changes that affect the build system or external dependencies (example scopes: gulp, broccoli, npm)
ci	Continuous Integration	Changes to the Continuous Integration (CI) configuration files and scripts (example scopes: Ansible, Travis, Circle, BrowserStack, SauceLabs)
docs	Documentation	Documentation only changes
feat	Features	A new feature
fix	Bug Fixes	A bug fix
perf	Performance Improvements	A code change that improves performance
refactor	Code Refactoring	A code change that neither fixes a bug nor adds a feature
style	Styles	Changes that do not affect the meaning of code (white-space, formatting, missing semi-colons, etc)
test	Tests	Adding missing tests or correcting existing tests

Table 8.4: Conventional commit types for Git. The most relevant are Features and Bug Fixes.

ID	Calculation	Description	Notation
T1	Bag of Words	Counting the total number of unique terms per document.	$f_{t,d}$
T2	Term Frequency	• $tf(t,d)$ denotes the number of unique terms per document, divided by the total number of terms • $t^{'}$ refers to a unique term • d refers to a document. A document refers to all unique commits in a branch in a repository.	$tf(t,d) = \frac{f_{t,d}}{\sum_{t^{'} \in d} f_{t^{'},d}}$
T3	Inverse Document Frequency	• $idf(t,D)$ is the logarithmically scaled inverse fraction of the documents that contain the term (obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient, where: • N refers to the corpus, the total number of documents. Also $N = \|D\|$. • $\|\{d \in D \div t \in d\}\|$ is the number of documents with term t. Also N_t. To prevent division by zero, the denominator is written as $(1 + N_t)$	$idf(t,D) = log\frac{N}{\|\{d \in D \div t \in d\}\|}$ $idf(t,D) = log\frac{N}{1+N_t}$
T4	Term Frequency-Inverse Document Frequency (TFIDF)	• $tdidf(t,d,D)$ refers to the number of documents d in corpus D a term t appears in, or the relevance of a word in a document related to all documents with that word. • t refers to the search term. • d refers to the documents. • D refers the number of documents in the corpus. • $tf(t,d)$ is the TF. See T2 for an explanation. • $idf(t,D)$ is the IDF. See T3 for an explanation.	$tdidf(t,d,D) = tf(t,d) \times idf(t,D)$

Table 8.5: Statistical methods for NLP and Text Mining.

A5. *Model.* A model in this context is defined as the layout of a set of layers, nodes and connections between the cells, including weights of connectors, summation function, and activation function. Different models, also referred to as 'neural network architecture', have different applications. E.g. a model for image recognition has another layout and other nodes and connections than a model for text processing [283]. Text is sequential data and documents and sentences can have different lengths.

A6. *Pretrained Model or Transfer Learning.*] With a pretrained model, experiences from previous training sessions with similar tasks can be used to speed up development. Bozinovski [284] already in 1976 introduced transfer learning. An example for NLP is BERT [285] that is in use at Google.

A7. Hyperparameter Settings. Hyperparameters are variables that are not part of the model but define how the model will operate. There is a trade-off between accuracy and speed of training and using the model based on the values of the hyperparameters. Typical settings are displayed in Table 8.6.

A8. *Finding Design Decisions in different Types of Information.* The objective of the types of information (Figure 8.2) is to convey an understanding for *why* decisions are taken for the specific software design.

Model Parameter	Hyperparameter
Internal to the model.	External to the model.
Value can be derived from data.	Value cannot be derived from data.
Estimated with historical data during training.	Manually set before training using heuristics from the practitioner.
Examples: • Weights; • Biases.	Examples of defining model architecture: • Number of hidden layers and hidden units; • Kernel size, stride, padding, pooling size. Examples of training optimization: • Learning rate; • Activation function; • Number of epochs; • Number and size of batches.

Table 8.6: Difference between model parameters and hyperparameters.

Different stakeholders –such as developer, end-user, customer, and manager– have different needs for information.

The third research question, 'Automated Text Analytics', is answered by this approach. The approach counts the eight most prominent artifacts: annotated data, model, hyperparameters, and transfer learning. The tool for source management control, i.e. Git, is most relevant because it is most close to the source without the need for other tools. Conditions for this approach are verbose Git comments. This approach is a new area of expertise to retrieve design decisions based on causal relations out of the text. It can be applied in all maturity levels where text is used. NLP is used for retrieving design decisions.

This approach, 'Automated Text Analytics', aims to reveal design decisions from existing documentation, particularly Git commit messages. This approach is most useful in phases when there are Git Commit messages in place but can be extended to all natural language media such as chats, mail, Confluence, and Jira.

8.4 Threats to Validity

For the previous studies ([20], [21]), the threats were addressed as follows. The initial search process identified threats concerning study selection, where the set of candidate papers for primary studies is selected, and the study filtering, where the final set of primary studies is determined. This threat was addressed by including the most used digital libraries in this area, which are also commonly used in secondary studies in software engineering. Typical examples are the selection of digital libraries, search string construction, and study selection bias. Threats concerning data validity were identified in the data extraction and analysis phases. Typical examples include data collection bias and publication bias. The risk of retrieving a small sample was mitigated by constructing a search string that could zoom in from a domain with over approximately 35.000 studies to about 200 relevant papers to answer the research questions. The threat of choosing the correct variables to be extracted was addressed through extensive discussions between the authors. The threat of publication bias (most identified primary studies coming from specific venues) was mitigated by snowballing. Furthermore, we addressed the threat of inadequate validity of primary studies through the inclusion criteria by only looking at peer-reviewed venues. Threats about research validity were identified over the whole mapping study and concerned the research design. Typical examples are generalizability and coverage of research questions. Extensive discussions among the authors mitigate the threat of the chosen research method bias, and the rationale of our decision is clearly described in the study design section. Furthermore, the authors have also discussed the choice and coverage of the research questions in multiple iterations. Regarding the generalizability of our results, they only apply within the scope of documentation in continuous software development.

Wieringa [11] defines the following threats to validity, applicable to this study:

1. Descriptive validity is the degree of support for a descriptive inference that refers to the accuracy, objectivity, and credibility of the information gathered. This threat is mitigated by using triangulation in methods and data. The methods are the a systematic mapping study, interviews, and case study. The data consist of literature, non-executable artifacts such as Git, the Atlassian[21] stack and other documents.

[21] https://www.atlassian.com/

RQ	Approach	Conditions	Characteristics	Artifacts
RQ1	Just Enough Upfront	Not specified.	Exploratory projects. Most applicable TRL: ≤3 Fit for Agile practices.	#: 16, of which most relevant: • Whiteboard Drawings, Sketches • Codified interface descriptions, • Plan of Approach, • Design decisions, • Accountability.
RQ2	Executable Documentation	'What' and 'why' must be well defined upfront.	Pipelines for CI/CD Applicable TRL: 4≤9 Fit for DevOps. Fast TTM.	#: 9, of which most relevant: • Frameworks, Templates, • TDD, BDD, • Infrastructure-as-code
RQ3	Automated Text Analytics	Verbose Git comments.	Rather new area of expertise. Applicable TRL: 1≤9 NLP for retrieving design decisions.	#: 8, of which most relevant: • Annotated data, Model, • Hyperparameter settings, • Transfer learning

Table 8.7: Answers to Research Questions.

2. Internal validity is the degree of support for explanations using causal relationships. This threat is mitigated by structuring the results from previous studies, defining sufficient and necessary conditions, and characteristics for the approaches.
3. External validity is the degree of support for the generalization of a theory so that is applicable in other domains then were the cases originate. This threat is not applicable because we focus on domain of software engineering and not other domains.
4. Construct validity is the degree to which inferences from phenomena to construct are justified.
5. Statistical conclusion validity is the degree of support for statistical inference. This threat is not applicable when defining approaches such as in this study.

8.5 Conclusions, Discussion and Future Research

Common to the research questions are requirements and characteristics for the three approaches. Documentation in CSD is about knowledge collection, knowledge building, and knowledge transfer to start ('what do I need to start?'), continue or deliver ('what do others need to continue?') a software product. With 'knowledge', we refer to all types of actionable information, including stakeholder concerns, requirements, specifications, source code, Git comments, end-user documentation, and values. People, including developers, tend to minimize the time and quality spent on documentation of a software product, so there is an urge to find novel ways to collect, transform and distribute knowledge.

In Figure 8.8 are the research questions displayed in relation to each other included with context. RQ1 applies to all phases but is more used in exploratory projects. For RQ2, requirements and specifications must be defined. RQ3 applies primarily to all phases where source-code is created or modified. The research questions for each phase relate to knowledge that is required upfront for the developer, knowledge that is build up while developing, and knowledge that is required afterwards by others to continue, use, operate and maintain. Table 8.7 shows the answers in brief to the research questions. A common characteristic concerns the acquisition, building, and transfer of knowledge.

To answer the first research question, the first approach to 'Just Enough Upfront' documentation to start was introduced. There are no specific requirements for this approach. Characteristics are that it applies more to exploratory projects where there are uncertainties about stakeholder concerns, technology, and process. This is typical for projects in the concept phase and a TRL lower than or equal to three. This approach is typical for projects where fast TTM is key to keeping up with competition or legislation. It is fit for Lean and Agile practices. There were sixteen artifacts identified, of which the most relevant are upfront: whiteboard sketches, codified interface description, and plan of approach. On delivery the documented design decisions and accountability artifacts were most relevant.

The second approach concerns 'Executable Documentation'. This approach requires well-defined projects, objectives, and targets to define specifications. The 'what' (requirements) and 'how' (specifications) must be well-described upfront. Typical for this approach are pipelines with CI/CD to achieve fast TTM. A typical process for this approach is DevOps. Infrastructure-as-code is typically part of the pipeline. There were nine artifacts identified, of which the most relevant are upfront: tests such as TDD and BDD. Furthermore, frameworks and templates are used upfront, and increments are added during development.

The third approach is about 'Automated Text Analytics'. A requirement is that Git comments are verbose and well-structured in English to retrieve helpful information. It is a relatively new area of expertise to use NLP for retrieving design decisions from git comments. The approach can be helpful for waterfall and iterative, incremental, processes. Contrary to the other approaches, the process method is not relevant. Eight artifacts

were identified, of which the most relevant are: annotated data, a model architecture, hyperparameter settings, and transfer learning.

8.5.1 Discussion of the Constructing of the Approaches

It is not possible to observe relations. This includes logical or statistical inferences. However, using cognition and mental models, relations can be validated in the case of inference methods such as deduction, induction, or statistical reasoning. There is no cognitive or mental model for causal inferences to validate these relations. One might infer a relation between identical events occurring after identical causes, but the relationship cannot be proven. This is a well-known problem in validating relations between observations but even harder when defining hypotheses or, in this study, approaches. The approaches are hypotheses that only in the weakest logical form, i.e. using abduction, can be formulated.

For the research paradigms, we consider two paradigms as relevant. At first, 'pragmatism' because abductive reasoning was introduced by Peirce, Houser, and Kloesel [223]. Second, 'constructivism' is the paradigm that March and Smith [243] consider to be applicable for design science.

8.5.2 Discussion of the Results: Novel Approaches Including Requirements, Characteristics, and Artifacts

The merits and applicability of artifacts including requirements and characteristics must be evaluated based on data from observations. The evaluation of approaches is typically tested in research methods such as case studies and data collection methods such as questionnaires, interviews, and applied statistics. See Figure 8.4 for the research methods and data collection methods. Causal relations cannot be observed. So, for validation, qualitative and quantitative data collection and analysis methods will be used that support or reject correlation.

8.5.3 Future Research

In future research, the proposed approaches will be evaluated. Candidate methods for evaluation are case studies, focus group studies and interviews. Through exploratory research, we already found indications for most of the artifacts. However, it requires further research to establish the necessary and sufficient requirements for which specific situations the approaches and artifacts are appropriate. For instance, additional textual communication such as chat conversations or mail probably contain design decisions as well.

An interesting avenue for future research involves the upfront condition for a codified interface description. In this study, it concerns the communication between sub systems. However, communication between people contributes to a better understanding for anything that is not documented or needs clarification. A communication protocol for team members becomes relevant in geographically distributed teams where team members have no face to face contact. So, the communication between people is of future interest.

Additionally, the third approach that describes the gap for automated analytics for natural language can be extended for reading whiteboard sketches by processing visual information. This third approach would then be extended with Automated Visual Analytics.

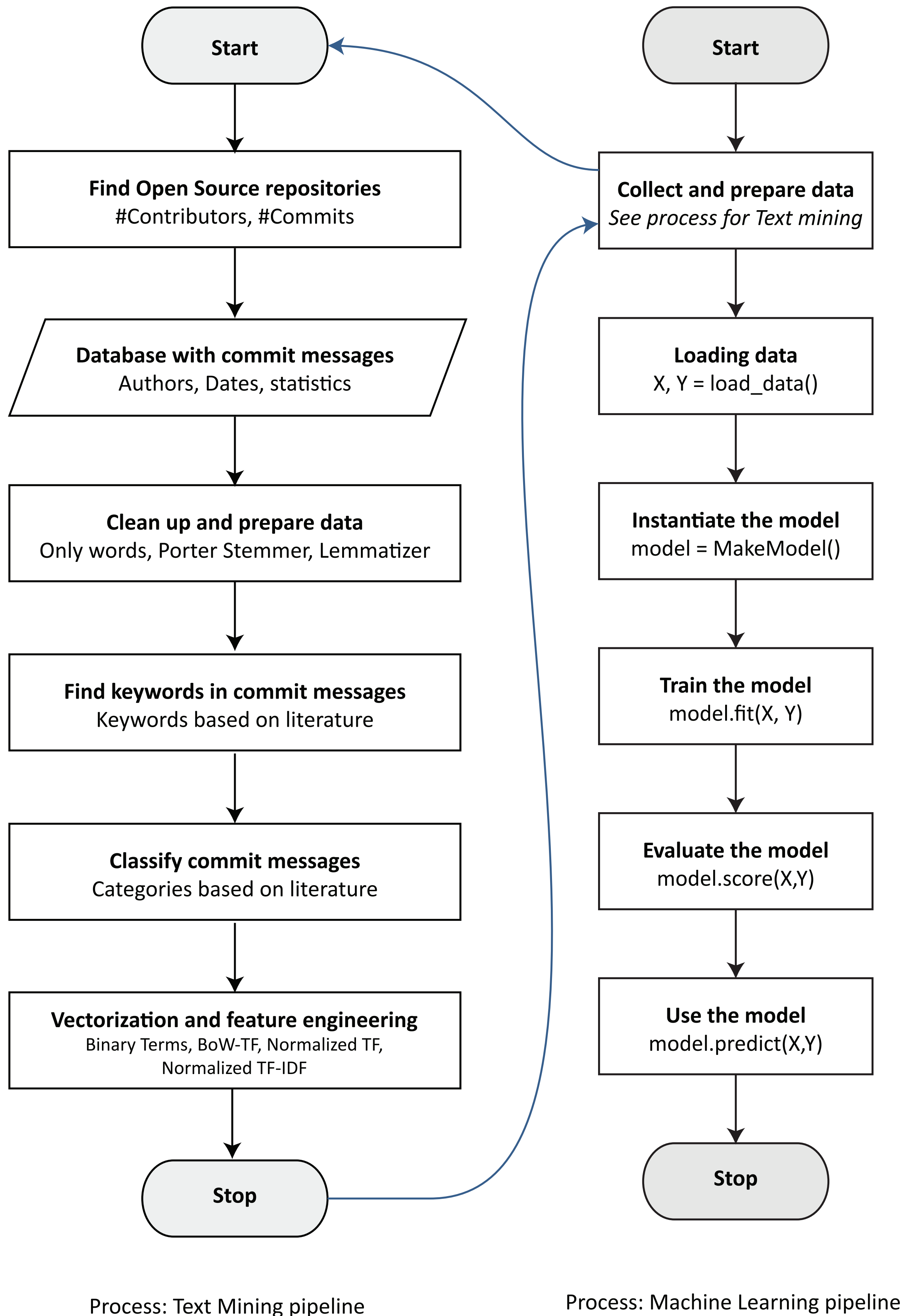


Figure 8.7: Pipelines for Text Mining and Machine Learning.

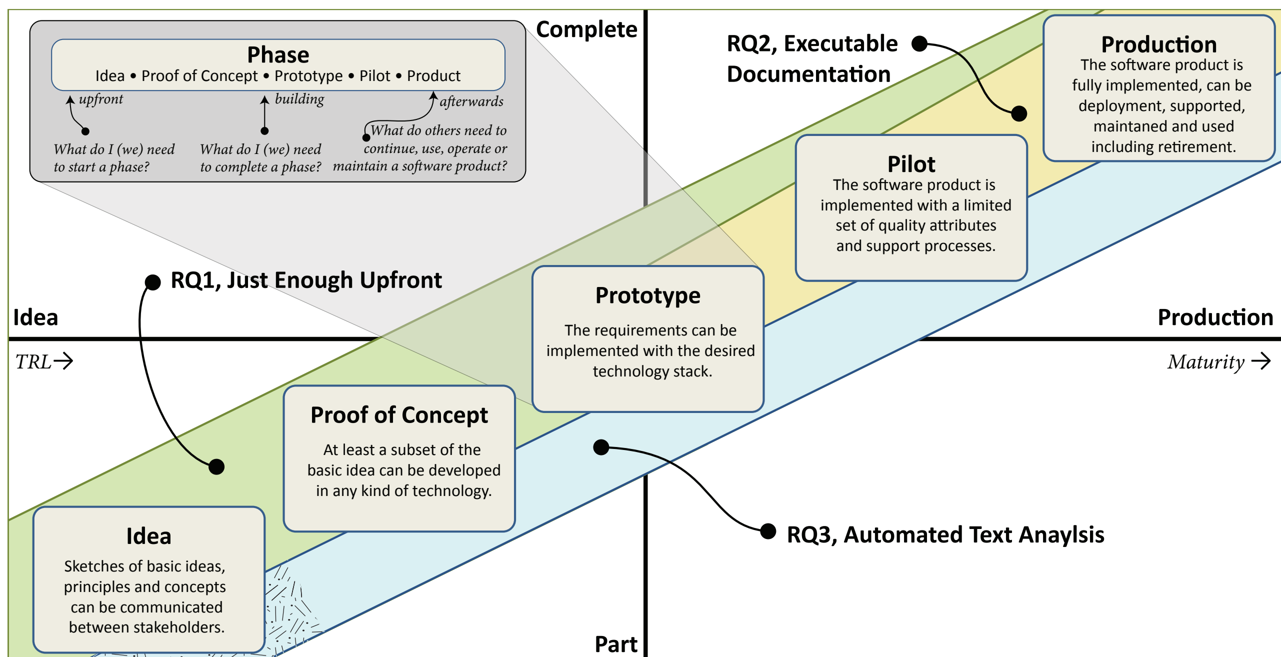


Figure 8.8: Relation between research questions, maturity, and completeness of the software product for each phase from Idea to Production.

Chapter 9

Evaluation of Approaches for Documentation in CSD[1]

Abstract With the adoption of values, principles, practices, tools and processes from Agile, Lean, and DevOps, knowledge preservation has become a serious issue because documentation is largely left out. We identify two questions that are relevant for knowledge acquisition and distribution concerning design decisions, rationales, or reasons for code change. The first concerns which knowledge is required upfront to start a project. The second question concerns continuation after initial development and addresses which knowledge is required by those who deploy, use or maintain a software product. We evaluate two relevant approaches for alleviating the issues, which are 'Just enough Upfront' and 'Executable Documentation', with a total of 25 related artifacts. For the evaluation, we conducted a case study supported by a literature review, organizational and project metrics, and a survey. We looked into closed source-code and closed classified source-code. We found two conclusive remarks. First, git commit messages typically contain what has been changed but not why source-code has been changed. Design decisions, rationale, or reasons for code change should be saved as close as possible to the source-code with Git Pull Requests. Second, knowledge about a software product is not only written down in artifacts but is also a social construction between team members. **Keywords** Artifacts, Continuous Software Development, Documentation, Executable Documentation, Just enough Upfront

9.1 Introduction

This study concerns the evaluation of novel approaches to documentation in CSD. CSD is an umbrella term that covers values, principles, practices, tools and processes from Agile, Lean, and DevOps [20]. Characteristics of CSD are that information about a software product is distributed across all tools which hold code and other (non)executable artifacts that stakeholders require to start an iteration or keep continuing an iteration. Another characteristic is that knowledge is loose, informal, and communicated in meetings such as daily stand-ups, leading to a risk of knowledge evaporation. In a previous study [19], we looked into information about software products, primarily to Git commit messages from open source repositories. In this study, we evaluate two approaches: 'Just enough Upfront' and 'Executable Documentation' concerning the characteristics of CSD. The first approach concerns 'just enough knowledge about stakeholder concerns, requirements, and specifications to start development'. The distribution of knowledge takes place through delivery of a design afterward, including design decisions. This approach is typically used for fast TTM situations. The second approach covers more mature projects where requirements and specifications are used at the start of TDD and

[1]This work was originally published as:

Theunissen, T., Hoppenbrouwers., S., & Overbeek., S. (2023). Evaluation of Approaches for Documentation in Continuous Software Development. *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, 404–411. doi:10.5220/0011846200003464

BDD.

To evaluate the approaches with artifacts, we studied practical usages in industry of artifacts with closed source code. 'Closed source code' refers to software which is not publicly available, including information about the software. Moreover, some code bases are also classified and are not even available to everyone within the organization studied. Special clearances were required, or developers had to be disconnected from the internet. Reasons were (national) security, privacy, or protecting (national) infrastructure against threats.

9.1.1 Research Project and Related Studies

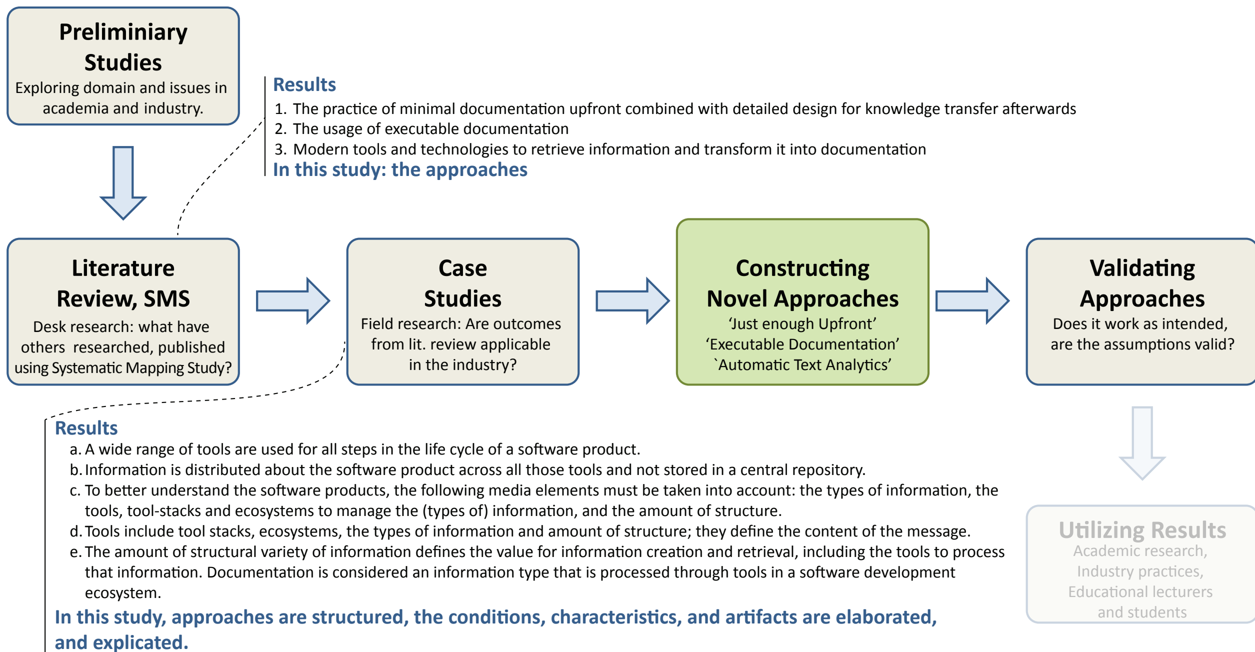


Figure 9.1: Phases in Research Project. This study concerns the evaluation, depicted in green.

This study concerns the evaluation of approaches to knowledge preservation in a research project that started with preliminary studies to explore the IT-engineering domain and knowledge preservation problems [30], [S31], [S32]. Following the preliminary study, a systematic mapping study was performed to find out what others have found already [20]. Following the literature review, the case studies in industry set out to validate the mapping studies and discover new issues, including the distribution of information about software products across a software development ecosystem [21]. In the construction phase, novel approaches such as 'Just enough Upfront' and 'Executable Documentation', including artifacts concerned, were described to cover the issues [19]. See Figure 9.1 for an overview.

In previous research, three novel approaches were constructed in response to the findings [19], of which two are evaluated and discussed in the current paper. These approaches are:

Just enough Upfront This approach leaves out big upfront design and encourages starting development as soon as possible. Agility in adapting to change is motivated by the progressive insight that leads to modified requirements and specifications, as inherent to CSD. There are no obstacles for applying this approach. Characteristics of the approach are that it is best used for exploratory projects. Exploration is typical for TRL ≤ 3 with PoC, but its exploration is also applicable to more mature phases such as prototyping, doing pilots, and taking software into production. The approach fits Agile and Lean practices. There are sixteen relevant artifacts, of which whiteboard sketches and a IDL plan are most relevant upfront. After delivery, design decisions and accountability are most applicable.

Executable Documentation This concerns any artifact related to a software product that is executable and is relatively easy to read by non-technical persons. Conditions are requirements (what), and specifications (how) that must be well-defined upfront. Characteristic for this approach is the use of pipelines in CI/CD [19]. The maturity level in TRL is $4 \leq 9$. It is suitable for DevOps and accommodates fast TTM. Of the nine relevant artifacts, frameworks, templates, libraries, and APIs are the most relevant. Typical processes are TDD and BDD. Developers and operators meet over infrastructure-as-code where concerns match, for instance for fast

TTM.

We found that Git commit messages typically contain *what* has been changed, and sometimes *how* the modification works. Documenting *what* has been changed is easily retrieved by looking at the differences between commits. Following the principle that Single Source of Truth (SSOT) conveys the ultimate truth [279], experienced developers and novice developers both can find out how the source code works by reading it. However, design decisions that document *why* the modification was made can hardly be found. Tools like Confluence and Jira are the tools mentioned by developers that record design decisions.

9.2 Research Design

9.2.1 Research Questions

RQ1 What are the necessary and sufficient conditions for evaluating our novel approaches of documentation in CSD?
This question concerns the research method to achieve rigor by demonstrating transparency and repeatability.

RQ2 For both approaches, which of the artifacts were used, are missing, or need adjustments of conditions and characteristics?
This question examines in detail the claims for merits.

RQ3 What are defining characteristics of obstacles that need to be resolved for implementing the novel approaches in industrial and educational context?
This question is a preparation for utilizing the approaches in actual usage.

9.2.2 Research Methods

In previous research, we used DS to construct and validate the novel approaches [11]. We will continue with DS for evaluating the approaches using case studies as proposed by Wieringa [11]. Case studies are used for data collection, following Yin [286]; see Figure 9.3. According to Wieringa [11, p. 31], validation concerns the assessment of potential usage of (in our case) novel approaches and their construction. Evaluation is the assessment of novel approaches, justified by assessing the actual outcomes compared with the intended outcomes. Assessment methods for validation are formative and summative judgments [287], [288]. Below, definitions for the judgments are given.

1. Theory evaluation includes formative validity and summative validity [288].
2. Formative validity refers to the process of building a new theory or approaches. A key characteristic is transparency [287].
3. Summative validity refers to the sum of the results of the theories or approach. It is achieved through artifact evaluation [287].

Figure 9.2 depicts the start, finish, process steps, and intermediate results. The intermediate results are explained in terms of data collection, analysis, and interpretation in the following sections.

9.3 Data Collection

We conducted nine case studies in five organizations with 28 units of analysis. One organization is commercial, one organization is educational, and three others are governmental. From five organizations, we consulted the participants with semi-structured interviews, studied non-executable artifacts such as documentation in tools (including Git commit messages), and reviewed source code. In Figure 9.3, the case studies are presented. In selecting the organizations, we targeted two specific approaches, i.e., exclusively 'Just enough Upfront' or 'Executable Documentation'.

9.3.1 A Myriad of Tools That Contain Information

The motivation for collecting this data is that it contributes to answering RQ2 and RQ3. A second motivation for collecting this data is that in previous research [21], we found that modern software development ecosystems include many tools that hold information about software products. This ranges from tools for capturing loose and informal communication, such as whiteboard sketches and natural language, to constructing source

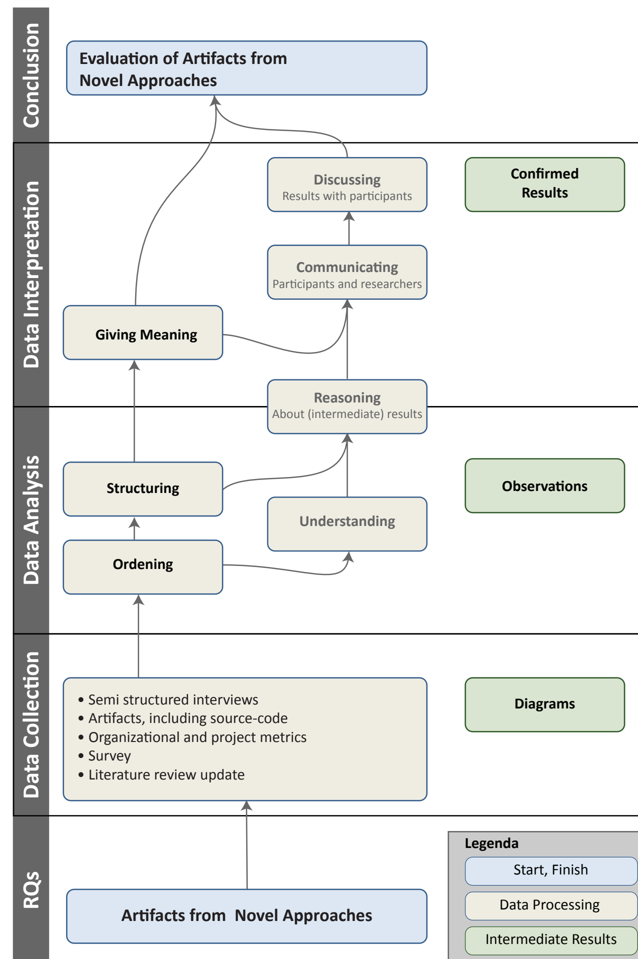


Figure 9.2: Study Design Process.

code and configuration data. For knowledge management, all organizations use Confluence, Jira, and Git. Confluence is the designated tool for all kinds of documentation, including design decisions. Jira is typically used for task and process management, and Git is used for source management control. See Figure 9.4 for the collected data.

9.3.2 Tenure of Team and Age of Repositories

The motivation for collecting this data is that it supports answering RQ2 and RQ3. A second motivation is that knowledge preservation becomes more relevant with applications aging because of the risk of knowledge vaporization. A team with senior developers working for years on an application or in the same organization shares embodied and tacit knowledge about values, principles, practices, and tools and processes, including changes over the years. See Figure 9.5 for the collected data.

9.3.3 Just Enough Upfront

The reason for collecting this data is that it contributes to answering RQ2 and RQ3. Furthermore, this data supports and suggests modifying conditions, characteristics, practices, and use cases for the previously [19] defined artifacts. See Figure 9.6 for the usage of the artifacts. Interviewees mentioned knowledge elicitation that compensates for missing artifacts, as an upfront activity in an educational context.

9.3.4 Executable Documentation

The reason for collecting this data is that it contributes to answering RQ2 and RQ3. Furthermore, this data supports and suggests modifying conditions, characteristics, practices and use cases for the previously [19]

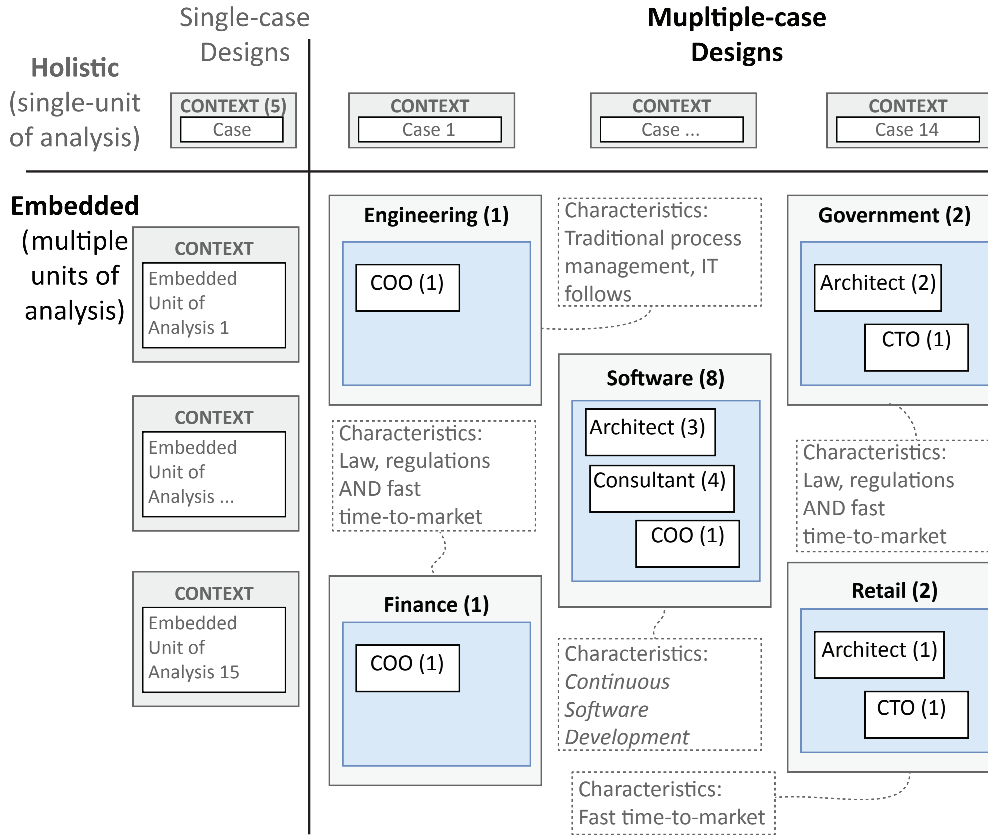


Figure 9.3: Units of Analysis in Case Studies.

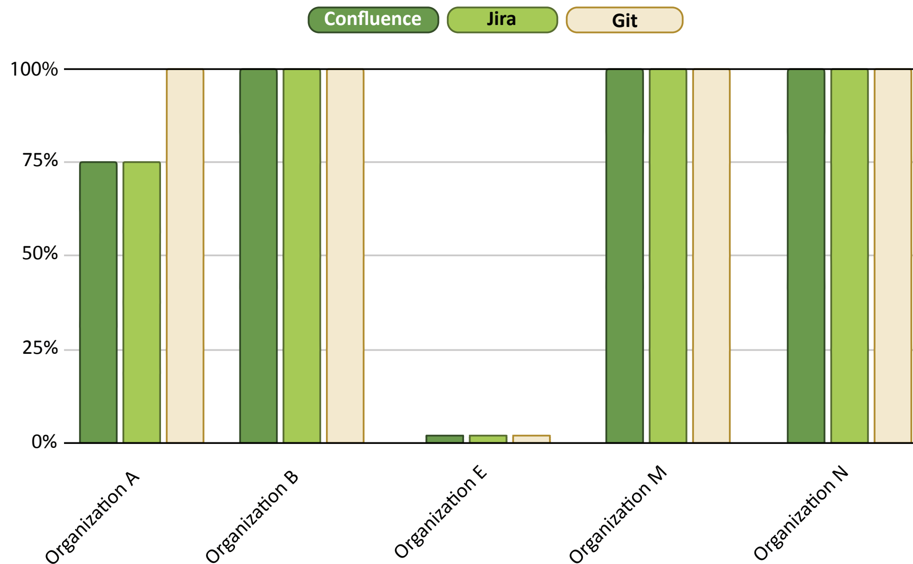


Figure 9.4: Usage of knowledge tools including design decisions. Git includes GitHub, GitLab, and Bitbucket

defined artifacts. See Figure 9.7 for the data. This data is skewed because it is not used in the educational organization.

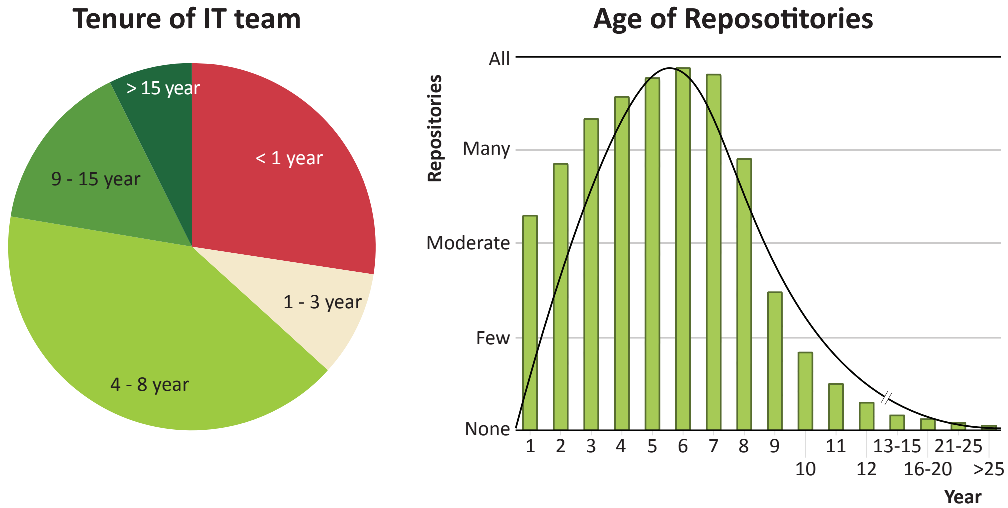


Figure 9.5: Tenure of team members for all organizations. Note that the diagram is skewed for < 1 year for the educational organization because students are only involved for one term (8-16 weeks). The scale of the right diagram is not proportional.

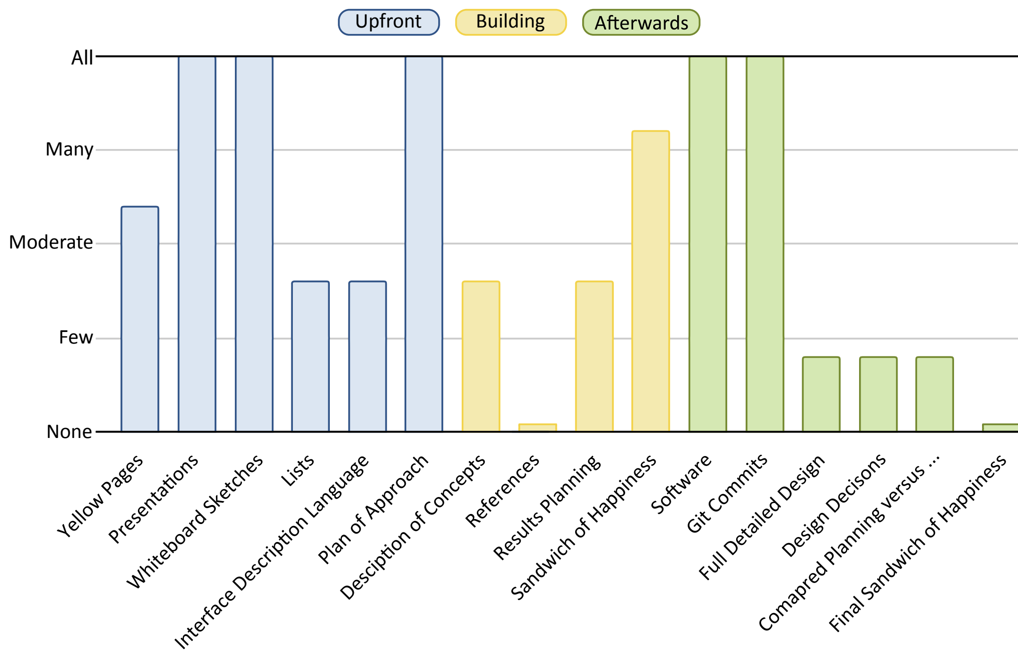


Figure 9.6: Usage of Artifacts for 'Just enough Upfront' across all organizations.

9.4 Data Analysis

Data analysis concerns the process of bringing order, structure, and meaning to the pile of collected data [289, p. 399]. Furthermore, remarkable results and omissions are mentioned. For this study, we assessed if the approaches and artifacts did have the intended outcomes. With this, we follow the summative assessment from DS.

9.4.1 A Myriad of Tools That Contain Information

Using a software development ecosystem entails that information about the software product is distributed across all tools in the ecosystem. Designated tools are used for specific types of information. For this research project, we are interested in design decisions. Only three tools were used for this type of information in all

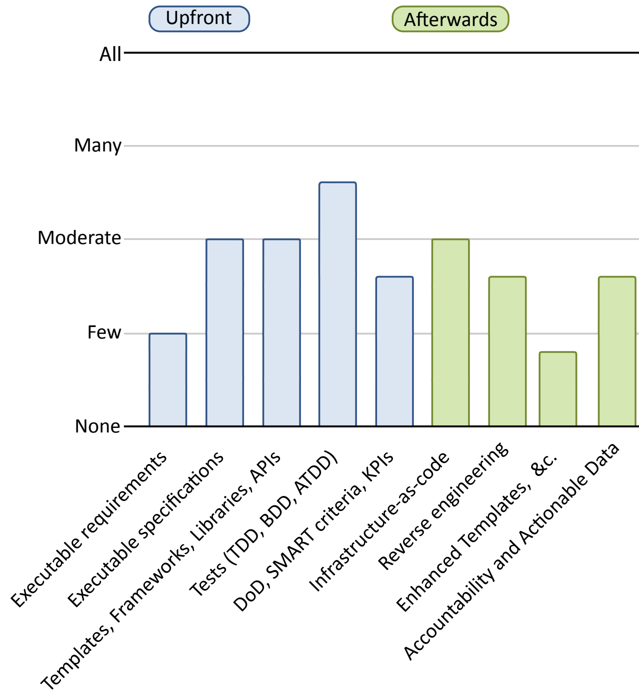


Figure 9.7: Usage of Artifacts for 'Executable Documentation' across all organizations.

organizations studied. These tools are Confluence, as a repository for knowledge about the software product, Jira, for task management, and Git, primarily for source code. For the range of types of information, varying from loosely communicated natural language and sketches on one side of the spectrum, to constructed source code that can compile to running applications on the other, design decisions are not stored in other tools. It may very well be possible that descriptions, interpretations and explanations as answers to questions might be stored in chats, emails or spoken language. However, we excluded these types of communication since they are not realistically retrievable. A typical workflow mentioned by the participants outlines the process as writing

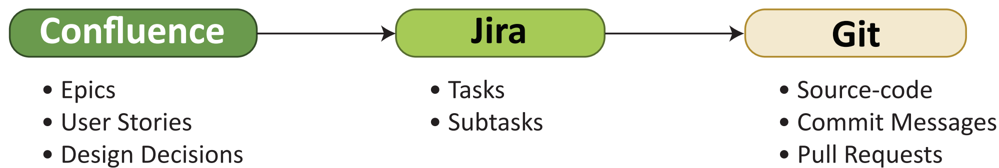


Figure 9.8: Typical workflow across all studied organizations.

epics and user stories in Confluence. Tasks from the user stories are managed with Jira, source code is stored in Git, and the comments can be limited to just the task number from Jira. Within this flow, code traceability from task to user story to epic should be guaranteed for knowledge preservation such as design decisions. See Figure 9.8. The flow matches with literature [20] and suggestions from Atlassian[2].

Observations from the case studies.

1. **No Confluence or Jira for running software products.** One of the organizations shuts down instances of Confluence and Jira at the end of a project. The end of a project does not entail the retirement of the software product, only that information is lost about the software product. For that organization, the Git commit messages only contain a task number, and the rationale for the code changes could not be retrieved. This leads to knowledge evaporation.
2. **Tooling without design decisions.** Another organization did not store design decisions and had to rely on a rather fuzzy vision without requirements or specifications for implementation. This organization

[2]https://www.atlassian.com/agile/project-management/user-stories

became aware of the lack of recorded design decisions and started to store these decisions in hindsight.
3. **Short lived projects.** For the educational organization, knowledge preservation is not relevant. The lifetime from start to retirement is only one term (8, 16 weeks). Students are required to learn to use standards like a SRS [290], SAD [90] or SDD [91]. Knowledge vaporization is immanent in this context.
4. **Migrations of tools over the years.** One organization has used software for dozens of years. Information about the software product must be available for over 40 to 50 years. Migration of tools did happen from paper to photocopies organized in a hierarchical file structure on disks to optical character recognition and eventually to Confluence.
5. **No design decisions in Git.** No organization mentioned keeping design decisions in Git. Discussing this made sense for participants, but it is no actual practice.

9.4.2 Tenure of Team and Age of Repositories

Compared with data from the U.S. [291], team members in the studied organizations keep working for the same organization much longer, except for the educational organization. This is a big difference compared to the tenures for developers in big tech companies, which is less than three years [292].

Observations from the case studies.
1. **Educational contexts require no history nor seniority.** Students are involved in a project for a short time, and maintenance or support is not part of the program. The teams investigated are larger than in the industry. The team in the educational organization consists of 20 to 25 developers and is divided into sub-teams working on sub-systems. The primary objective for students is to pass the course and project, not to maintain or support a software product. Students remember tools, techniques, and processes rather than design decisions.
2. **Team with history in an organization - Sharing knowledge.** One of the participants explained that employees in non-profit organizations earn less money than in commercial companies but are more loyal to the organization. Some organizations have a network of family members.
   Employees stay much longer in the three governmental organizations than in commercial organizations. Team members have a history of past decisions, which is relevant because not all decisions are documented. This positively affects knowledge preservation, even if it is not documented. The memory of undocumented historic decisions is not identified as an artifact in the literature.
3. **Life span of software.** The moderate age of software repositories is six years, with a Gaussian distribution ranging from 1 year to 12 years [293]. The distribution is skewed with mode and median left from the mean because newer software is in use more than older software. The software can age for dozens of years for one of the studied organizations. This software is part of cyber-physical systems such as frigates and submarines.

9.4.3 Just Enough Upfront

Observations from the case studies:
1. **Artifacts always used.** Some artifacts were always used, for example presentations, source code, and commit messages. Presentations are helpful in transferring knowledge that can serve as input for a project or iteration because the mix of high-level diagrams supports understanding of causal and logical relations between concepts. Depth can be obtained by supporting text accompanying the diagrams [248]. Source code serves as a single source of truth and reveals what the software product does and how it works through close reading by (experienced) developers. It does not explain *why* the software product works. Tools like Confluence or Jira are often used to record the reasons for modifications. For Git, individual commit messages were not valuable, as mentioned by several respondents. However, Pull Requests are in use that “should contain design decisions” instead of ordinary commit messages.
2. **Artifacts never used.** Some artifacts, such as references and a final SoH, were never used. The artifacts SoH and RP are not familiar in most organizations, but assessments and retrospection are common across all organizations.
3. **Knowledge distributed across software development ecosystem.** In previous studies, we found that nowadays, many tools are in use that store pieces of information about software products [21] which remains true. However, design decisions, rationales, and reasons for change are typically stored in tools like Confluence or Jira.
4. **Design decisions keeping close to source code.** Source code is the SSOT. Developers can read the source code and understand *what* it does and *how* it works. However, the reason *why* the source code

is as it is cannot be retrieved from the source code itself. Typical tools in use are Confluence and Jira. We incorrectly expected that design decisions were stored as close as possible to source code, and not in other tools. To emphasize this point: one organization decided to turn off Confluence and Jira because the project was operational. As a result, the knowledge related to the project was gone.

9.4.4 Executable Documentation

Observations from the case studies:

1. **Artifacts always used.**There are no artifacts concerning ED that are always used across all organizations.
2. **Artifacts never used.** ED is not used in many organizations, see Figure 9.7. Only a few organizations practice it. An exception concerns TDD. The highest score is because participants prefer other tests, especially unit tests. This type of testing often concerns the 'happy flow'. TDD aims at writing tests that fail. In the case studies, one organization used ED for all projects in their division. This organization did not use any other documentation for knowledge acquisition or knowledge transfer. This implies a lack of testing of the software product. Developers mentioned reasons such as test cases that made it possible to reverse-engineer software that should be kept secret. Another organization with long-term software products (> 30 years) relies solely on its developers. Developers are not connected to the internet to 'google' an answer or use publicly available libraries and frameworks for security reasons.

9.5 Data Interpretation

In this section, the most remarkable results will be presented, being either a confirmation or a rejection of proposed artifacts. Artifacts such as presentations, whiteboard sketches, software, and git commits are used by all participants. Furthermore, following RQ3, defining characteristics for obstacles to implementation will be mentioned.

1. **Relax on design upfront.** Developers can start a project or iteration as soon as requirements and specifications are sufficient. Longer contemplation does not prevent progressive insights, especially in an educational context where progressive insights are implicit in the process and objectives.
   No obstacle hinders the implementation of relaxing on upfront design. Relaxing does not imply becoming sloppy. The effort starts with remembering values, principles, practices, and tools and processes to make proper judgments on requirements and priorities.
2. **Strict on codified interface description.** Integration of (sub)systems is always hard, it also happens at the end (when deadlines are approaching), and the blame falls on others. Starting with, and holding on to, a codified interface prevents integration issues.
   Barriers to integration are related to being accountable for results one has no control over: the other (sub)systems, teams or external parties. This leads to short-sighted vision, losing the big picture and shared responsibilities.
3. **Knowledge is social.** Not all knowledge is stored in artifacts. Participants across all organizations mentioned that design decisions, rationales, or reasons for change are not documented. For some organizations, where security, reliability, or confidentially are critical, knowledge about the software product is in the hearts and minds of senior developers. Knowledge is shared in meetings, conversations, or presentations. People are loyal to their organization, and knowledge stays within the organization. This supports knowledge preservation.
   There is a remarkable distinction between open source code, closed source code, and closed classified source code. Open source has the best quality of documentation. Closed source code has the poorest quality, with sometimes only a Jira task number. Closed classified source relies on a loyal team of senior team members to answer questions from new team members.
   A risk for accepting that knowledge could be found in social interaction concerns teams with a high change rate of team members, including changes caused by internal reorganizations. The organizations we researched do not pay the highest salaries, but team members have been loyal to the organization over the years.
4. **Knowledge is primarily *NOT* distributed across a software development ecosystem.** Information about the software product is distributed across all tools in a software development ecosystem in an organization. However, some types of information, such as design decisions, rationales, or reasons for change, are kept in a single place.
   There is no barrier to this result. A communicated *modus operandi* supports knowledge preservation.

5. **Saving design decisions in Git.** It is common practice across all organizations to use Git (GitHub, GitLab, Bitbucket) for keeping source code. Typically, commit messages do not add much information about the source code changes as they mention *what* has been changed or *how* it works. As design decisions cannot be retrieved from source code, the best option for keeping this type of information are git commit messages. Some participants mention that Pull Requests are the designated commit types. We could not identify a hurdle for keeping design decisions as close to the source code as possible. Participants could not give a clear reason why not to do so. Speculation from the participants includes the way of working and tooling. The way of working is a behavioral change, and tooling is a management decision.

9.6 Threats to Validity

Threats to validity in DS is not a mature discipline [294]. We follow Gonzalez and Sol [288] with formative and summative assessments to evaluate the treatments. Formative assessment concerns the process of how a result is achieved, and summative assessment refers to the result ('does it work?') of the treatment. Wieringa [11, pp. 128, 138] mentions the following threats for treatment design:

1. *Inference support.* What are reasonings or statistical schemes to draw valid conclusions based on assumptions and observations?
2. *Repeatability.* Can data sampling and reasoning be reproduced several times, leading to the same conclusions by other researchers?
3. *Ethics.* Does the research harm participants?
4. *Interpretation.* Does the reader accept the interpretation as a fact?

9.6.1 Inference Support

This risk is mitigated by the verification of assumptions and hypotheses with the participants. An *assumption* is a statement that is considered to be valid and proven in an inference of taking to be invalid, which leads to an invalid conclusion by applying the principle of excluded middle. A *hypothesis* is considered to be true when it is not possible to falsify it. Some artifacts, see Figures 9.6 and 9.7 were valid (value 'All') and some were not valid (value 'None'), and some were inconclusive (other values than 'None' or 'All').

For this threat, both formative and summative assessments are relevant. The formative assessment considers the validity of the reasoning process. The summative assessment concerns the validity of assumptions and hypotheses.

9.6.2 Repeatability

This risk is partially mitigated. For the data sampling, it is mitigated in principle. However, because of the classified source code of some organizations, clearance level is required[3] or legal requirements[4] are required.

For this threat, summative assessments are relevant. With this assessment, the repeatability is evaluated as to whether results can be reproduced, partially when a situation is changed, or can not be reproduced.

9.6.3 Ethics

Internal processes mitigate this risk in organizations by excluding privacy-sensitive data from the research. Some organizations have a legal task to use violence (military) or detect acts of crime. This could harm people (enemies, criminals) involved but not society as such. We take a utilitarian ethical viewpoint that values the happiness of society above the happiness of the individual.

We consider the formative assessment applicable where the process is evaluated.

[3]'Verklaring omtrent Gedrag' (VOG, Certificate of Conduct) or 'Verklaring van geen bezwaar' (VGB, Certificate of no Objection).

[4]'Wet op de Openbaarheid van Bestuur' (WOB, 'Dutch Freedom of Information Act').

9.6.4 Interpretation

This risk is mitigated by method and data triangulation. We used a literature review, case studies, and a survey to have several methods. Furthermore, we used semi-structured interviews, executable and non-executable artifacts, and data from a survey.

For this threat, both formative and summative assessments are applicable. The formative assessment concerns the process of collecting and interpreting the data. For the summative assessment, results might differ because of progressing insight, whereas a different result might emerge with identical assumptions and hypotheses.

9.7 Conclusions and Future Work

We have three research questions to answer.

The first research question concerns the necessary and sufficient conditions to use approaches and artifacts. Shared values, principles, practices, tools, and processes are critical for necessary and sufficient conditions.

The second research question assesses the actual usage of approaches and artifacts. 'Just enough Upfront' is an approach that is used across all organizations, or is considered appealing. Some artifacts are constantly in use: presentations, whiteboard diagrams, plans of approach, software, and commit messages. Some are never used, such as references and final SoH. For the approach 'Executable Documentation', no conclusive artifacts are included or excluded. The artifacts that were most in use are tests.

The third research question concerns the defining characteristics of barriers to implementing the approaches and artifacts. An obstacle mentioned across all organizations concerns unconfirmed, loose deviations from prescribed processes, or interfering objectives. Examples of loose deviations of prescribed processes are left-outs of ceremonies of textbook definitions of Scrum or Scaled Agile Framework (SAFe).

Conclusive remarks concern one observation and one consideration. The observation is about the social construction of knowledge, where knowledge is not a mere act of intellect or rational or intelligible epistemic contemplation. The consideration concerns design decisions, rationales, or reasons for change that should be saved as close as possible to the source code in Git. What and how of the source code can be read in the code. A rationale cannot be retrieved from the source code. A separation of source code and design decisions does not contribute to knowledge preservation.

9.7.1 Future Work

Future work concerns the implementation of the outcomes in an educational and a professional context. Specifically, the 'Just enough Upfront' approach is a candidate for implementation, including the most used artifacts. Secondly, an effort must made to save design decisions with Git Pull Requests.

Chapter 10

Conclusion

10.1 Research Questions

The main objective and research question of this research project is:

> *What are the necessary and sufficient conditions for effective communication with just enough documentation in CSD obtaining insight and control to start building, delivering, maintaining, and continuously using a software product?*

Conditions are an arrangement for a specific situation, as in a hypothesis or conjecture, that is applicable. Next, there is a difference between necessary and sufficient conditions, which logically and causally relate to each other. Necessary conditions refer to minimal requirements for an event to occur, although not sufficient. An example of a necessary condition is a minimum of documentation that is required for deployment, usage, or maintenance. In contrast, extensive documentation can still be insufficient because incomprehensible to understand what, how, or why. Sufficient conditions concern satisfactory requirements for an event to occur, although not necessary. For example, knowledge preservation can be obtained by multiple types of representation, such as text, diagrams, or personal explanations. None of these is exclusively required. Effective communication refers to the receiver of the information who acts as intended by the sender. The expression 'just enough' applies to the bare minimum of information for obtaining results.

Writers and readers need to have common ground for communication in order to understand and agree on the essentials of objectives, requirements, and specifications for the software product involved. This requires clear and concise communication focusing on those essentials.

The objective is investigated through three research questions.

RQ1 Why is knowledge acquisition, building, preserving, and revealing in CSD, leading to knowledge vaporization, a hard problem?

We found the following issues that make knowledge hard to acquire, build, and distribute: Developers often prioritize coding over tasks such as documentation, meetings, presentations, or other activities that hinder their productivity in producing working code. However, this leads to a dispersion of knowledge about the software product across multiple tools in the development ecosystem, lacking a central repository. Consequently, team members do not store or share design decisions, rationales, and reasons for change. Also, incomplete or inaccurate interface descriptions further complicate the integration of subsystems. These challenges can be attributed to the Agile Manifesto's emphasis on working software over comprehensive documentation, categorizing non-value-adding activities as waste. Similarly, achieving fast TTM through infrastructure as code takes precedence in DevOps. As a result, developers often use these principles as excuses to forgo investing time in knowledge acquisition and distribution. However, effective team communication becomes increasingly difficult when dealing with factors such as rapid changes, geographical dispersion, diverse time zones, and cultural or language barriers. Establishing a common understanding of values, priorities, and knowledge becomes essential for clear and concise communication among team members.

Next, epistemological questions about the nature of knowledge and the relation between the mind and reality make it even more difficult. This classical problem has been researched for the last 2500 years without conclusive answers. Additionally, there has been a cultural shift towards shorter attention spans

ID	Phase	Process	Artifact
D1	Upfront	Communication between stakeholders	• Yellow Pages.
D2		Shaping thoughts	• Presentation.
D3			• Whiteboard and drawings.
D4			• Lists.
D5		Communication between systems	• Interface Description Language.
D6		Planning	• Plan of Approach.
D7	Building	Progressive Insights	• Description of Concepts.
D8		Communication between systems	• References.
D9		Coaching and Control	• Results Planning.
D10			• Sandwich of Happiness.
D11	Afterwards	Deliverables	• Software.
D12			• Git Comments.
D13			• Full Detailed Design.
D14			• Decisions.
D15		Accountability	• Compared Planning versus Actual Results.
D16			• Final Sandwich of Happiness.

Table 10.1: Phases, Processes, and Artifacts for Approach 'Just Enough Upfront'.

and a different way of processing information among those who have grown up with digital technology, known as 'digital natives', compared to 'traditional' paper book readers.

RQ2 What are defining and contextual characteristics for CSD?

We use CSD as an umbrella term that is characterized by:

1. Values, principles, practices, tools, and processes from Agile, Lean, and DevOps;
2. Fast TTM because of market demands, competition, technological innovations, or legal requirements;
3. Using CI/CD for continuous testing, integration, and automated delivery;
4. Challenges such as a complex multitude of interconnected systems, external third party systems that are out of control, or a high degree of uncertainty;
5. Activities in the complete life cycle of a software product, from concept to end-of-life;
6. The software development ecosystem with a variety of tools from conceptualizing to monitoring production systems;
7. Distributed information about a software product across all tools in a software development ecosystem. There is no central repository that stores all information;
8. The continuously changing state of a software product due to external factors, new features, bugs, and progressive insights;
9. Involvement of web applications, mobile applications, and enterprise systems for communication between back-end and front-end;

RQ3 Which documentation artifacts are required: a) *upfront* for *you* to start a project or an iteration, b) *while building with team members* the software product, and c) *afterward by others* for deployment, maintenance, and usage -in short: continuation- of a software product?

First, we have found that the aspect of continuity, besides software development, also applies to continuous learning. To accommodate continuous learning, we introduced two complementary methods: the first is RP, and the second the SoH. The RP assesses achieved results, whether successful or not. Activities are defined as smartish results, one significant, relevant result per team member per part of a day, i.e., the morning or the afternoon. So, a team of five members has ten results per week. Over time, results are getting more relevant and contribute significantly more to the end product. The SoH is a reflection and introspection on the process of the production of the results. There are three questions to answer: 1) Looking backward, what was successful, were contributions, lessons learned, or are you proud of; in short: what was good? 2) Looking backward, what were failures, irritations, frustrations, and misguided ambitions, should be avoided; in short: what was bad? 3) Looking forward, show adaptability for continuous change, the balance between overestimated competencies and underestimated difficulties, and balance between anxiety and boredom, defined by smartish KPIs.

Furthermore, for continuous learning, two methods were described. These are the RP and SoH. The proposed RP is planning divided into significant, relevant results per half a day per team member, e.g., a team of five members defines ten significant, relevant results per week. It includes an assessment of obtained results, whether successful or not including. Accountability is a link to a commit hash, updated results when successful, or a brief explanation for failure. The RP is complemented with a SoH.

ID	Phase	Process	Artifact
E1	Upfront	Defining *what* has to be done.	• Executable Requirements.
E2		Defining *how* it has to be done.	• Executable Specifications.
E3		Using knowledge from previous experiences.	•Templates, Frameworks, Libraries, APIs.
E4		Quality control.	•Tests (TDD, BDD, ATDD).
E5		Management.	• Definition of Done, Acceptance Criteria, SMART KPIs.
E6	Afterwards	Speeding up the CI/CD cycle.	• Infrastructure-as-code.
E7		Retrieving knowledge about software products.	• Reverse Engineering.
E8		Saving executable knowledge for future development.	• Enhanced Templates, Frameworks, Libraries, APIs.
E9		Accountability	• Accountability and Actionable Data.

Table 10.2: Phases, Processes, and Artifacts for Approach 'Executable Documentation'.

ID	Phase	Process	Artifact
A1	Upfront	Source Code	• Git Comments.
A2	Building	Text Mining	• Search Categories and Search Terms.
A3			• Statistics: BoW, TF, IDF, TFIDF.
A4			• Annotated data.
A5		Deep Learning	• Model.
A6			• Pretrained Model or Transfer Learning.
A7			• Hyperparameter Settings.
A8	Afterwards		• Finding Design Decisions in Different Types of Information.

Table 10.3: Phases, Processes, and Artifacts for Approach 'Automated Text Analysis'.

10.2 Contributions

This research contributes to three communities: academia, industry, and education.

10.2.1 Academic Researchers

The primary contribution concerns approaches with artifacts for documentation used in acquiring knowledge upfront and the distribution of knowledge after the delivery of the software product. The approaches are 'Just Enough Upfront' with sixteen artifacts, 'Executable Documentation' with nine artifacts, and 'Automated Text Analysis' with eight artifacts. Table 10.4 shows the key characteristics

We identified conditions and characteristics for the approaches. Characteristics concern the maturity from idea to production. Approaches change during the life cycle of a software project while requirements, the technology of choice, and processes get better defined. Although discovering architecture decisions after the software product has been released is not new, see, for example: [295], [296]. Our approach, 'Automated Text Analysis', adds two aspects. The first is saving design decisions, rationale, or reasons for a change in git commit messages and pull requests. The second aspect is identifying causal relations in text with NLP to reveal architecture decisions made possible by recent developments in Artificial Neural Network (ANN). Furthermore, we found that information about a software product is distributed across many tools in a software development ecosystem instead of stored in one single repository. We showed that specific types of information are stored in specific tools at specific places, such as design decisions in Git.

Approach	Conditions	Characteristics	Artifacts
Just Enough Upfront	Not specified.	Exploratory projects. Most applicable TRL: $\leq$3 Fit for Agile practices.	#: 16, of which most relevant: • Whiteboard Drawings, Sketches • Codified interface descriptions, • Plan of Approach, • Design decisions, • Accountability.
Executable Documentation	'What' and 'why' must be well defined upfront.	Pipelines for CI/CD Applicable TRL: 4$\leq$9 Fit for DevOps. Fast TTM.	#: 9, of which most relevant: • Frameworks, Templates, • TDD, BDD, • Infrastructure-as-code
Automated Text Analytics	Verbose Git comments.	Rather new area of expertise. Applicable TRL: 1$\leq$9 NLP for retrieving design decisions.	#: 8, of which most relevant: • Annotated data, Model, • Hyperparameter settings, • Transfer learning

Table 10.4: Key Conditions, Characteristics, and Artifacts per Approach.

With developments in Large Language Model (LLM)s since December 2022, such as ChatGPT[1], outlooks are promising for identifying logical and causal relations in text pointing to design decisions, rationale, or reasons for a change in software. The availability of a small version of LLM with Llama[2] from Meta[297] makes it possible to run pre-trained models on laptops. However, as has been found in our research, results depend on verbose git comments for revealing meaningful design decisions.

10.2.2 Practitioners in the Industry

We found confirmation of the TL;DR habit. At the same time, practitioners underlined the urge for knowledge preservation. Dynamics in the industry demand fast TTM with multiple drivers, including technological innovations, legal requirements, competition, bugs, and new features. We found that the approach 'Just Enough Upfront' is sufficient to start developing if stakeholders have only a blurry view of objectives, requirements, and specifications as long as it can be communicated in a presentation. There is not always a need for formalized UML documents. A second aspect requires more attention than typically is paid. This concerns a codified interface description of the communication between (sub)systems. For projects that are characterized by well-defined requirements, proven technology, and CI/CD pipelines, the approach with 'Executable Documentation' assists in knowledge preservation and fast TTM.

10.2.3 Teachers and Students in Education

The most important contribution for students is using progressive insights in projects. From a didactic point of view, spending a significant amount of time learning how to do knowledge elicitation, defining requirements, and next specifying details using templates and UML is a good start. The same applies to defining a plan of approach. For students, the dynamics of fast TTM do not apply, but progressive insights and making errors are part of a learning process. The approach 'Just Enough Upfront' defines artifacts for this approach. Last but not least, the approaches and artifacts save time for both students and teachers to be more efficient for 'Just Enough Upfront' and 'Executable Documentation'.

Teachers and students benefit from the aforementioned approaches in an educational context. Students do not like to read and write, and the same applies to teachers who have to read many documents on reflection and knowledge preservation of students.

1. Start with writing down *why* changes are made in source-code for junior students. Use Git commit messages for documenting design decisions. Students better understand stakeholder concerns, algorithms, and end-user feedback if they describe the reasons for starting, building, or retiring changes in projects.
2. Relax on big upfront design. In an educational context, progressive insights are welcomed.
3. Relax on following a plan of approach. In an educational context, progressive insights are welcomed. This implies a change of objectives, approaches, requirements, specifications, or implementations. Following a plan in an educational context shows a lack of progressive insights.
4. The interface description of communication between (sub)systems must be stricter. Integration of (sub)systems is difficult, and an issue in the industry often involves multiple providers. Learning to think from a detailed view to an abstract overview and from a contextual view of lines of code ensures an optimal match for issues and implementation.
5. Explore ANN for software engineers, especially RNN for text analysis to reveal design decisions from text, as opposed to CNN typically used for image data.

10.3 Future Research and Utilizing Results

Utilization of the results of the study in the communities.

10.3.1 Future Research in Academia

Further research on 'Automated Text Analysis' to reveal architectural design decisions, rationale, and reasons for change by identifying causal relations in text using NLP. The scope of Git commit messages and pull

[1] https://chat.openai.com
[2] https://github.com/ggerganov/llama.cpp

requests can easily be extended to other textual documents stored in other tools such as Confluence or chat messages. Another option includes exploring other information types with less structure, such as whiteboard sketches or saved online video meetings. Furthermore, analyzing types of information with a lesser amount of structure.

10.3.2 Implementation in Industry

For the industry, the two methods concerning RP and SoH are relevant for CSD. Next, the three approaches with 33 artifacts are all applicable in the industry, although maybe not at the same time and not in all situations. To be specific, implementation of results in the industry starts with saving design decisions, rationale, and reasons for change as close to the source as possible, that is, the source-code as in Git where the commit messages and pull requests contain always *why* a change is saved. Practitioners should codify an interface strictly so that no confusion or misunderstanding arises when programming (sub)systems.

10.3.3 Implementation in Education

An artifact that showed effective knowledge preservation is using Git to store design decisions, rationale, and reasons for a change instead of commenting on what has been changed or how the change affects existing code.

The following items are more value related than the approaches and artifacts 'Just Enough Upfront', 'Executable Documentation', and 'Automated Text Analysis' mentioned in respectively Table 10.1, Table 10.2, and Table 10.3.

1. Embrace progressive insights is one of the essential parts of learning. Learning methods such as UML. SRS, SAD or SDD has an edifying goal. However, planning and thinking ahead can be considered complete if enough information is at hand to start developing. This implies relaxing on big upfront design and on strictly following a plan of approach.
2. Lessons learned, learning to fail, and learning from errors and failures contribute to a learning process. This applies specifically to education, where lessons learned are the core of the activities.
3. Emphasize personal qualities and preferences. Some students excel in exploring new concepts, can deal with uncertainties, and are aware of context, while others feel comfortable understanding instructions, following procedures, and contributing to efficiency. There is no good or bad related to these qualities. Both are required in the life cycle of a software product. The SoH emphasize continuous growth in individual values by emphasizing introspection and reflection on individual contributions to accomplishments, both successes, and failures.
4. Learning is both an individual and a social activity. Learning is not so much a product of a rational, cognitive, individual activity but as much a social construct, including communication on values and facts. Typical individual activities that are reasons for staying at home instead of coming to class are reading, practicing, and understanding. Typical reasons to come to class are working together, presenting, and discussing.

For students and teachers, one of the key concepts for learning is progressive insights, including learning from mistakes to make better mistakes next time. A detailed UML document upfront contributes only a little to understanding the issues and gravity of difficulties. It takes a relatively large amount of time that does not lead to a better understanding or prevention of errors. Short iterations with assessments like the 'Sandwich of Happiness' contribute to both assessing results and providing insights into how the results were achieved.

Bibliography

[1] K. Beck, M. Beedle, A. Van Bennekum, *et al.* “Manifesto for Agile Software Development Twelve Principles of Agile Software”, Manifesto for Agile Software Development. (2001), [Online]. Available: https://agilemanifesto.org/.

[2] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit.* Boston, MA: Addison-Wesley Educational, 2003, ISBN: 0-321-15078-3.

[3] L. Bass, I. Weber, and L. Zhu, *Devops: A Software Architect’s Perspective*, 1st. Addison-Wesley Professional, 2015, ISBN: 0-13-404984-5.

[4] M. T. Cicero, E. W. Sutton, and H. Rackham, *Cicero: In Twenty-eight Volumes. 3: De Oratore: Books I-III* (The Loeb Classical Library 348), Reprinted. Cambridge, Mass.: Havard Univ. Press, 2001, ISBN: 978-0-674-99383-9.

[5] C. E. Shannon and W. Weaver, “The Mathematical Theory of Communication”, *Urbana: University of Illinois Press*, 1949.

[6] M. Pawlowski, T. Paterek, D. Kaszlikowski, V. Scarani, A. Winter, and M. Zukowski, “Information Causality as a Physical Principle”, *Nature*, vol. 461, no. 7267, pp. 1101–1104, Oct. 2009. DOI: 10.1038/nature08400. [Online]. Available: http://arxiv.org/abs/0905.2292 (visited on 01/30/2021).

[7] P. Naur and B. Randell, “Software Engineering: Report of a Conference Sponsored by the NATO Science Committee”, NATO, Garmisch, Germany, NATO, 1969.

[8] D. L. Parnas, *Information Distribution Aspects of Design Methodology.* Pittsburgh, Pa.: Carnegie Mellon University, Dept. of Computer Science, 1971, ISBN: 0-7204-2063-6.

[9] R. Overton, “Developments in Computer Aided Software Maintenance”, AMS Inc Claremont CA, Technical Report, 1974.

[10] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2. [Online]. Available: http://link.springer.com/10.1007/978-3-642-29044-2 (visited on 01/30/2021).

[11] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ISBN: 978-3-662-43838-1. [Online]. Available: https://doi.org/10.1007/978-3-662-43839-8 (visited on 01/30/2021).

[12] P. Kruchten, H. Obbink, and J. Stafford, "The Past, Present, and Future for Software Architecture", *IEEE software*, vol. 23, no. 2, pp. 22–30, 2006.

[13] M. Shaw and P. Clements, "The Golden Age of Software Architecture", *IEEE software*, vol. 23, no. 2, pp. 31–39, 2006.

[14] K. D. Maxwell and P. Forselius, "Benchmarking Software Development Productivity", *IEEE Software*, vol. 17, no. 1, pp. 80–88, 2000.

[15] R. Hoda, N. Salleh, and J. Grundy, "The Rise and Evolution of Agile Software Development", *IEEE software*, vol. 35, no. 5, pp. 58–63, 2018.

[16] G. S. Walia and J. C. Carver, "A Systematic Literature Review to Identify and Classify Software Requirement Errors", *Information and Software Technology*, vol. 51, no. 7, pp. 1087–1109, Jul. 2009, ISSN: 09505849. DOI: 10.1016/j.infsof.2009.01.004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584909000111 (visited on 03/31/2023).

[17] R. N. Longuemare and J. M. Lodal, "Pentagon Agrees to Release Source Codes on Case-by-Case Basis", *Inside the Pentagon*, vol. 13, no. 16, pp. 1–22, 1997, ISSN: 2164-814X. [Online]. Available: https://www.jstor.org/stable/43993619 (visited on 03/12/2023).

[18] C. f. D. a. R. Health. "Policy for Device Software Functions and Mobile Medical Applications", U.S. Food and Drug Administration. (), [Online]. Available: https://www.fda.gov/regulatory-information/search-fda-guidance-documents/policy-device-software-functions-and-mobile-medical-applications (visited on 03/12/2023).

[19] T. Theunissen, S. Hoppenbrouwers, and S. Overbeek, "Approaches for Documentation in Continuous Software Development", *Complex Systems Informatics and Modeling Quarterly (CSIMQ)*, vol. 32, pp. 1–27, 2022, ISSN: 2255-9922. DOI: 10.7250/csimq.2022-32.01.

[20] T. Theunissen, U. van Heesch, and P. Avgeriou, "A Mapping Study on Documentation in Continuous Software Development", *Information and Software Technology*, vol. 142, p. 106 733, 2022, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106733. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058492100183X.

[21] T. Theunissen, S. Hoppenbrouwers, and S. Overbeek, "In Continuous Software Development, Tools Are the Message for Documentation", in *Proceedings of the 23th International Conference on Enterprise Information Systems*, J. Filipe, M. Smialek, A. Brodsky, and S. Hammoudi, Eds., SCITEPRESS - Science and Technology Publications, 2021, ISBN: 978-989-758-509-8. DOI: 10.5220/0010367901530164.

[22] M. Steup and R. Neta, "Epistemology", in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Fall 2020, Metaphysics Research Lab, Stanford University, 2020. [Online]. Available: https://plato.stanford.edu/archives/fall2020/entries/epistemology/ (visited on 03/05/2023).

[23] K. R. Subramanian, "Myth and Mystery of Shrinking Attention Span", *International Journal of Trend in Research and Development*, vol. 5, no. 3, 2018. [Online]. Available: http://www.ijtrd.com/papers/IJTRD16531.pdf.

[24] F. Jabr, "The Reading Brain in the Digital Age: The Science of Paper Versus Screens", *Scientific American*, vol. 11, no. 5, 2013. [Online]. Available: https://www.scientificamerican.com/article/reading-paper-screens/.

[25] A. N. Whitehead, "Process and Reality: An Essay in Cosmology [1929]", in *Science*, D. R. Griffin and D. W. Sherburne, Eds., Simon and Schuster, 1978, ISBN: 978-0-02-934570-2.

[26] S. M. Koosel, "TL;DR: Temporality Shifts in Digital Culture", *AoIR Selected Papers of Internet Research*, 2015. [Online]. Available: https://spir.aoir.org/ojs/index.php/spir/article/download/9073/7164.

[27] C. Su, H. Zhou, L. Gong, B. Teng, F. Geng, and Y. Hu, "Viewing Personalized Video Clips Recommended by Tiktok Activates Default Mode Network and Ventral Tegmental Area", *NeuroImage*, vol. 237, p. 118 136, 2021. DOI: 10.1016/j.neuroimage.2021.118136.

[28] D. Rothman, *A Tsunami of Learners Called Generation Z*, 2016. [Online]. Available: https://mdle.net/Journal/A_Tsunami_of_Learners_Called_Generation_Z.pdf.

[29] K. Popper, *Conjectures and Refutations: The Growth of Scientific Knowledge.* Routledge; 2nd edition (August 9, 2002), 2014, ISBN: 978-0-415-28594-0.

[30] T. Theunissen and U. van Heesch, "The Disappearance of Technical Specifications in Web and Mobile Applications", in *Software Architecture*, ser. Software Architecture. ECSA 2016. Lecture Notes in Computer Science, B. Tekinerdogan and U. Zdun, Eds., vol. 9839, Springer International Publishing, 2016, pp. 265–273. DOI: 10.1007/978-3-319-48992-6_20. [Online]. Available: https://doi.org/10.1007/978-3-319-48992-6_20.

[33] T. Theunissen, S. Overbeek, and S. Hoppenbrouwers, "Continuous Learning with the Sandwich of Happiness and Result Planning", in *26th European Conference on Pattern Languages of Programs*, ser. EuroPLoP'21, New York, NY, USA: Association for Computing Machinery, 2021, ISBN: 978-1-4503-8997-6. DOI: 10.1145/3489449.3489974.

[34] U. van Heesch and P. Avgeriou, "Mature Architecting - a Survey About the Reasoning Process of Professional Architects", in *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society, 2011, pp. 260–269. DOI: 10.1109/WICSA.2011.42.

[35] M. Ciolkowski, O. Laitenberger, S. Vegas, and S. Biffl, "Practical Experiences in the Design and Conduct of Surveys in Empirical Software Engineering", *Empirical Methods and Studies in Software Engineering*, pp. 104–128, 2003. DOI: 10.1007/978-3-540-45143-3_7.

[36] C. Teddlie and F. Yu, "Mixed Methods Sampling a Typology with Examples", *Journal of mixed methods research*, vol. 1, no. 1, pp. 77–100, 2007. DOI: 10.1177/1558689806292430.

[37] S. Sonnenburg, "Creativity in Communication: A Theoretical Framework for Collaborative Product Creation", *Creativity and Innovation Management*, vol. 13, no. 4, pp. 254–262, 2004. DOI: 10.1111/j.0963-1690.2004.00314.x.

[38] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf, "A Characterization Framework for Software Deployment Technologies", Colorado State Univ Fort Collins Dept of Computer Science, Technical Report, 1998. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA452086.

[39] D. J. Teece, "Capturing Value from Knowledge Assets: The New Economy, Markets for Know-How, and Intangible Assets", *California Management Review*, vol. 40, no. 3, pp. 55–79, 1998. DOI: 10.2307/41165943.

[40] J. F. Rayport and J. J. Sviokla, "Exploiting the Virtual Value Chain", *Harvard Business Review*, vol. 73, no. 6, p. 75, 1995. [Online]. Available: https://hbr.org/1995/11/exploiting-the-virtual-value-chain.

[41] R. A. Howard, "Information Value Theory", *Systems Science and Cybernetics, IEEE Transactions on*, vol. 2, no. 1, pp. 22–26, 1966. DOI: 10.1109/TSSC.1966.300074.

[42] T. Tamai and Y. Torimitsu, "Software Lifetime and Its Evolution Process Over Generations", in *Software Maintenance, 1992. Proceerdings., Conference On*, IEEE, 1992, pp. 63–69. DOI: 10.1109/ICSM.1992.242557. [Online]. Available: https://ieeexplore.ieee.org/document/242557.

[43] M. Armbrust, A. Fox, R. Griffith, *et al.*, "A View of Cloud Computing", *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010. DOI: 10.1145/1721654.1721672. [Online]. Available: https://dl.acm.org/doi/10.1145/1721654.1721672.

[44] T. Eriksson and J. Ortega, "The Adoption of Job Rotation: Testing the Theories", *Industrial & Labor Relations Review*, vol. 59, no. 4, pp. 653–666, 2006. [Online]. Available: https://journals.sagepub.com/doi/pdf/10.1177/001979390605900407.

[45] Bureau of Labor Statistics, U.S. Department of Labor. "Employee Tenure in 2012: The Economics Daily: U.S. Bureau of Labor Statistics". (2022), [Online]. Available: https://www.bls.gov/opub/ted/2012/ted_20120920.htm (visited on 09/11/2022).

[46] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, *et al.*, "Preliminary Guidelines for Empirical Research in Software Engineering", *IIEEE Trans. Software Eng.*, vol. 28, no. 8, pp. 721–734, Aug. 2002, ISSN: 00985589. DOI: 10.1109/TSE.2002.1027796. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1027796.

[47] M. Mason, "Sample Size and Saturation in PhD Studies Using Qualitative Interviews", *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, vol. 11, no. 3, Aug. 2010. DOI: 10.17169/fqs-11.3.1428. [Online]. Available: https://www.qualitative-research.net/index.php/fqs/article/view/1428.

[48] F. Erich, C. Amrit, and M. Daneva, "Report: DevOps Literature Review", University of Twente, Tech. Rep, Technical Report, 2014. [Online]. Available: http://rgdoi.net/10.13140/2.1.5125.1201 (visited on 01/30/2021).

[49] S. Alliance. "What Is Scrum? An Agile Framework for Completing Complex Projects-Scrum Alliance", What Is Scrum? (2017), [Online]. Available: https://www.scrumalliance.org/.

[50] B. Kent and A. Andres, *Extreme Programming Explained: Embrace Change, Second Edition.* Addison Wesley Professional, 2004, ISBN: 0-321-27865-8. [Online]. Available: http://proquestcombo.safaribooksonline.com/0321278658?tocview=true.

[51] R. Wilsenach. "DevOps Culture", Saatavissa (viitattu 23.4. 216): (2016), [Online]. Available: https://martinfowler.com/bliki/DevOpsCulture.html.

[52] M. Walls, *Building a DevOps Culture.* O'Reilly Media, Inc., 2013, ISBN: 978-1-4493-6836-4.

[53] J. Humble and J. Molesky, "Why Enterprises Must Adopt DevOps to Enable Continuous Delivery", *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011. [Online]. Available: https://www.scientificamerican.com/article/reading-paper-screens/.

[54] C. Lovelock and E. Gummesson, "Whither Services Marketing? In Search of a New Paradigm and Fresh Perspectives", *Journal of service research*, vol. 7, no. 1, pp. 20–41, 2004. DOI: 10.1177/10946705042661.

[55] D. Cukier, "Devops Patterns to Scale Web Applications Using Cloud Services", in *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, ACM, 2013, pp. 143–152. DOI: 10.1145/2508075.2508432.

[56] F. P. Brooks Jr, *The Mythical Man-Month (Anniversary Ed.)* Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-83595-9.

[57] D. Dunning, "The Dunning-Kruger Effect. on Being Ignorant of One's Own Ignorance", in *Advances in Experimental Social Psychology*, M. P. Zannia, P. Devine, J. M. Olson, and A. Plant, Eds., 1st ed., vol. 44, Elsevier Inc., 2011, pp. 247–296, ISBN: 978-0-12-385522-0. [Online]. Available: http://dx.doi.org/10.1016/B978-0-12-385522-0.00005-6.

[58] Atlassian. "Visualise Your Roadmap", Products & News. (2017), [Online]. Available: https://www.atlassian.com/blog/archives/visualize-your-roadmap.

[59] D. North *et al.* "Introducing BDD", Better Software, March. (2006), [Online]. Available: https://dannorth.net/2006/10/20/article-introducing-behaviour-driven-development/.

[62] U. van Heesch, P. Avgeriou, and R. Hilliard, "Forces on Architecture Decisions - a Viewpoint", in *Proceedings of the 2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture : 20-24 August 2012, Helsinki, Finland*, M. A. Babar, T. Männistö, C. E. Cuesta, and J. E. Savolainen, Eds., IEEE, 2012. DOI: 10.1109/WICSA-ECSA.212.18.

[63] G. Fairbanks, *Just Enough Software Architecture: A Risk-Driven Approach.* 2010, ISBN: 978-0-9846181-0-1.

[64] P. Kruchten, P. Lago, and H. van Vliet, "Building up and Reasoning About Architectural Knowledge", in *Quality of Software Architectures*, ser. Lecture Notes in Computer Science, C. Hofmeister, I. Crnkovic, and R. Reussner, Eds., vol. 4214, Springer Publishing Company, 2006, pp. 43–58, ISBN: 978-3-540-48820-0.

[65] O. Zimmermann, "Microservices Tenets: Agile Approach to Service Development and Deployment", *Comput Sci Res Dev*, vol. 32, no. 3-4, pp. 301–310, Jul. 2017. DOI: 10.1007/s00450-016-0337-0. [Online]. Available: http://link.springer.com/10.1007/s00450-016-0337-0 (visited on 01/30/2021).

[66] R. Soley *et al.*, "Model Driven Architecture", *OMG white paper*, vol. 308, no. 308, p. 5, 2000. DOI: 10.1007/978-3-540-45242-3_2.

[67] J. D. Herbsleb and D. Moitra, "Global Software Development", *IEEE software*, vol. 18, no. 2, pp. 16–20, 2001. DOI: 10.1109/52.914732.

[68] R. C. Martin, *Clean Code-Refactoring, Patterns, Testen Und Techniken Für Sauberen Code: Deutsche Ausgabe.* MITP-Verlags GmbH & Co. KG, 2013, ISBN: 978-3-8266-5548-7.

[69] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture", *IEEE Softw.*, vol. 22, no. 2, pp. 19–27, Mar. 2005, ISSN: 0740-7459. DOI: 10.1109/MS.2005.27. [Online]. Available: http://ieeexplore.ieee.org/document/1407822/ (visited on 01/30/2021).

[70] U. Zdun, R. Capilla, H. Tran, and O. Zimmermann, "Sustainable Architectural Design Decisions", *IEEE Softw.*, vol. 30, no. 6, pp. 46–53, Nov. 2013. DOI: 10.1109/MS.2013.97. [Online]. Available: http://dx.doi.org/10.1109/MS.2013.97.

[71] M. Nowak, C. Pautasso, and O. Zimmermann, "Architectural Decision Modeling with Reuse: Challenges and Opportunities", in *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, 2010, pp. 13–20. DOI: 10.1145/1833335.1833338.

[72] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, 2003, ISBN: 978-0-13-597444-5.

[73] M. Hüttermann, *DevOps for Developers* (The Expert's Voice in Web Development), M. Hüttermann, Ed. New York: Apress : Distributed to the book trade worldwide by Springer Science+Business Media New York, 2012, ISBN: 978-1-4302-4569-8.

[77] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions", in *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society, 2005, pp. 109–120, ISBN: 0-7695-2548-2. DOI: 10.1109/WICSA.2005.61.

[78] A. H. Mohamed, "Capturing Software-Engineering Tacit Knowledge", in *Proceedings of the 2nd Conference on European Computing Conference (ECC'08:)*, 2008. DOI: 10.5555/1895260.1895271.

[79] G. Borrego, A. L. Morán, R. R. Palacio, A. Vizcaíno, and F. O. García, "Towards a Reduction in Architectural Knowledge Vaporization During Agile Global Software Development", *Information and Software Technology*, vol. 112, pp. 68–82, 2019. DOI: 10.1016/j.infsof.2019.04.008.

[80] P. Lago and H. Van Vliet, "Explicit Assumptions Enrich Architectural Models", in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, IEEE, 2005, pp. 206–214. DOI: 10.1109/ICSE.2005.1553563.

[81] A. Bazaz, J. D. Arthur, and J. G. Tront, "Modeling Security Vulnerabilities: A Constraints and Assumptions Perspective", in *2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, IEEE, 2006, pp. 95–102. DOI: 10.1109/DASC.2006.35.

[82] I. Habli and T. Kelly, "Capturing and Replaying Architectural Knowledge Through Derivational Analogy", in *Second Workshop on Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent (SHARK/ADI'07: ICSE Workshops 2007)*, IEEE, 2007, pp. 4–4. DOI: 10.1109/SHARK-ADI.2007.6.

[83] B. Dagenais, H. Ossher, R. K. Bellamy, M. P. Robillard, and J. P. De Vries, "Moving into a New Software Project Landscape", in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 275–284. DOI: 10.1145/1806799.1806842.

[84] B. Fitzgerald, K.-J. Stol, R. O'Sullivan, and D. O'Brien, "Scaling Agile Methods to Regulated Environments: An Industry Case Study", in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 863–872. DOI: 10.1109/ICSE.2013.6606635.

[85] C. Ebert and M. Paasivaara, "Scaling Agile", *IEEE Software*, vol. 34, no. 6, pp. 98–103, 2017. DOI: 10.1109/MS.2017.4121226.

[87] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley Professional, 2000, ISBN: 978-0-321-60580-1.

[88] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America, “A General Model of Software Architecture Design Derived from Five Industrial Approaches”, *Journal of Systems and Software*, vol. 80, no. 1, pp. 106–126, Jan. 2007, ISSN: 01641212. DOI: 10.1016/j.jss.2006.05.024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121206001634 (visited on 01/30/2021).

[90] Technical Committee, “ISO/IEC/IEEE 42010:2011 - Systems and Software Engineering — Architecture Description”, Joint Technical Committee ISO/IEC JTC 1, Geneva, Switzerland, ISO/IEC/IEEE, 2011. [Online]. Available: https://www.iso.org/standard/50508.html.

[91] Standards Committee, “IEEE Std 1016-2009 (Revision of IEEE Std 1016-1998), IEEE Standard for Information Technology—Systems Design—Software Design Descriptions”, Joint Technical Committee ISO/IEC JTC 1, Geneva, Switzerland, Standard, 2009. DOI: 10.1109/IEEESTD.2009.5167255. [Online]. Available: https://ieeexplore.ieee.org/iel5/5167253/5167254/05167255.pdf.

[92] A. Tang, P. Liang, and H. Van Vliet, “Software Architecture Documentation: The Road Ahead”, in *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, IEEE, 2011, pp. 252–255. [Online]. Available: http://www.cs.rug.nl/search/uploads/Publications/tang2011sad.pdf.

[93] B. A. Kitchenham, D. Budgen, and O. P. Brereton, “Using Mapping Studies as the Basis for Further Research–a Participant-Observer Case Study”, *Information and Software Technology*, vol. 53, no. 6, pp. 638–651, 2011. DOI: 10.1016/j.infsof.2010.12.011.

[94] K. Petersen, S. Vakkalanka, and L. Kuzniarz, “Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update”, in *Information and Software Technology*, vol. 64, Aug. 2015, pp. 1–18. DOI: 10.1016/j.infsof.2015.03.007. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584915000646.

[95] V. R. Basili, G. Caldiera, and H. D. Rombach, “The Goal Question Metric Approach”, *Encyclopedia of software engineering*, vol. 2, pp. 528–532, 1994. DOI: 10.1.1.104.8626. [Online]. Available: http://maisqual.squoring.com/wiki/index.php/The%20Goal%20Question%20Metric%20Approach.

[96] P. Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley, 2004, ISBN: 978-0-321-19770-2.

[98] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, *et al.*, “Continuous deployment of software intensive products and services: A systematic mapping study”, *Journal of Systems and Software*, vol. 123, pp. 263–291, Jan. 2017, ISSN: 01641212. DOI: 10.1016/j.jss.2015.12.015. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121215002812 (visited on 01/30/2021).

[99] P. Diebold and M. Dahlem, “Agile Practices in Practice: A Mapping Study”, in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE ’14*, London, England, United Kingdom: ACM Press, 2014, pp. 1–10. DOI: 10.1145/2601248.2601254. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2601248.2601254 (visited on 01/30/2021).

[100] V. T. Heikkilä, D. Damian, C. Lassenius, and M. Paasivaara, “A Mapping Study on Requirements Engineering in Agile Software Development”, in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, Madeira, Portugal: IEEE, 2015, pp. 199–207. DOI: 10.1109/SEAA.2015.70. [Online]. Available: http://ieeexplore.ieee.org/document/7302452/.

[101] K. Curcio, T. Navarro, A. Malucelli, and S. Reinehr, “Requirements Engineering: A Systematic Mapping Study in Agile Software Development”, *Journal of Systems and Software*, vol. 139, pp. 32–50, May 2018, ISSN: 01641212. DOI: 10.1016/j.jss.2018.01.036. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121218300141 (visited on 01/30/2021).

[102] M. Shafiq and U. sman Waheed, "Documentation in Agile Development A Comparative Analysis", in *2018 IEEE 21st International Multi-Topic Conference (INMIC)*, IEEE, Karachi: IEEE, Nov. 2018, pp. 1–8, ISBN: 978-1-5386-7536-6. DOI: 10.1109/INMIC.2018.8595625. [Online]. Available: https://ieeexplore.ieee.org/document/8595625/ (visited on 01/30/2021).

[103] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic Mapping Studies in Software Engineering", *12th International Conference on Evaluation and Assessment in Software Engineering*, vol. 17, p. 10, Jun. 1, 2008, ISSN: 02181940. DOI: 10.14236/ewic/EASE2008.8.

[104] B. Kitchenham, P. Brereton, M. Turner, *et al.*, "The Impact of Limited Search Procedures for Systematic Literature Reviews — a Participant-Observer Case Study", in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, A. S. I. of Electrical and E. Engineers, Eds., IEEE Computer Society, 2009, pp. 336–345. DOI: 10.1109/ESEM.2009.5314238.

[105] B. A. Kitchenham, P. Brereton, M. Turner, *et al.*, "Refining the Systematic Literature Review Process—Two Participant-Observer Case Studies", *Empir Software Eng*, vol. 15, no. 6, pp. 618–653, Dec. 2010. DOI: 10.1007/s10664-010-9134-8. [Online]. Available: http://link.springer.com/10.1007/s10664-010-9134-8 (visited on 01/30/2021).

[106] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from Applying the Systematic Literature Review Process Within the Software Engineering Domain", *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, Apr. 2007. DOI: 10.1016/j.jss.2006.07.009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016412120600197X (visited on 01/30/2021).

[107] C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering", *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, pp. 1–10, 2014. DOI: 10.1145/2601248.2601268. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2601248.2601268 (visited on 01/30/2021).

[108] H. Zhang, M. A. Babar, and P. Tell, "Identifying Relevant Studies in Software Engineering", *Information and Software Technology*, vol. 53, no. 6, pp. 625–637, Jun. 2011. DOI: 10.1016/j.infsof.2010.12.010. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584910002260 (visited on 01/30/2021).

[109] W. B. Cavnar, J. M. Trenkle, *et al.*, "N-Gram-Based Text Categorization", in *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, Citeseer, vol. 161175, 1994. [Online]. Available: https://www.let.rug.nl/~vannoord/TextCat/textcat.pdf.

[110] T. A. Schwandt, *Qualitative Data Analysis: An Expanded Sourcebook.* Thousand Oaks, CA, US: Sage Publications, Inc, 1994, vol. 19, ISBN: 0-8039-4653-8. DOI: 10.1016/0149-7189(96)88232-2.

[113] A. Rüping, *Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects.* Hoboken, NJ: John Wiley & Sons, 2004, vol. 51, ISBN: 978-0-470-85617-8. [Online]. Available: http://www.drdobbs.com/architecture-and-design/agile-documentation-strategies/197003363%20%7B%%7D5Cn.

[132] M. Lopez-Nores, J. J. Pazos-Arias, J. Garcia-Duque, *et al.*, "Bringing the Agile Philosophy to Formal Specification Settings", *Int. J. Soft. Eng. Knowl. Eng.*, vol. 16, no. 6, pp. 951–986, Dec. 2006, ISSN: 0218-1940. DOI: 10.1142/S0218194006003075. [Online]. Available: https://www.dropbox.com/home/publications?preview=lpeznores2006.pdf.

[147] M. O. Ahmad, D. Dennehy, K. Conboy, and M. Oivo, "Kanban in Software Engineering: A Systematic Mapping Study", *Journal of Systems and Software*, vol. 137, pp. 96–113, Mar. 2018, ISSN: 01641212. DOI: 10.1016/j.jss.2017.11.045. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0164121217302820.

[162] R. Wieringa, N. Maiden, N. Mead, and C. Rolland, "Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion", *Requirements Eng*, vol. 11, no. 1, pp. 102–107, Mar. 2006. DOI: 10.1007/s00766-005-0021-6. [Online]. Available: http://link.springer.com/10.1007/s00766-005-0021-6 (visited on 01/30/2021).

[169] C. Le Goues and S. Yoo, “Search-Based Software Engineering 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings”, in *Conference Proceedings SSBSE*, Springer, 2014, p. 164. DOI: 10.1007/978-3-319-09940-8. [Online]. Available: https://link.springer.com/content/pdf/10.1007/978-3-319-09940-8.pdf.

[170] T. Keuler, S. Wagner, and B. Winkler, “Architecture-Aware Programming in Agile Environments”, in *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012*, IEEE, Helsinki, Finland: IEEE, 2012, pp. 229–233, ISBN: 978-0-7695-4827-2. DOI: 10.1109/WICSA-ECSA.212.35. [Online]. Available: http://ieeexplore.ieee.org/document/6337725/.

[174] M. Kajko-Mattsson, “Problems in Agile Trenches”, in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM ’08*, ser. ESEM ’08, New York, NY, USA: ACM, 2008, p. 111, ISBN: 978-1-59593-971-5. DOI: 10.1145/1414004.1414025. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414025%20http://portal.acm.org/citation.cfm?doid=1414004.1414025%20https://www.dropbox.com/home/publications?preview=kajkomattsson2008.pdf.

[175] M. Cohn, *User Stories Applied: For Agile Software Development* (Addison-Wesley Signature Series). Boston: Addison-Wesley, 2004, ISBN: 978-0-321-20568-1.

[189] I. D. Coman and G. Succi, “An Exploratory Study of Developers’ Toolbox in an Agile Team”, in *Agile Processes in Software Engineering and Extreme Programming*, P. Abrahamsson, M. Marchesi, and F. Maurer, Eds., vol. 31, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 43–52, ISBN: 978-3-642-01852-7. [Online]. Available: http://link.springer.com/10.1007/978-3-642-01853-4_7 (visited on 01/30/2021).

[201] J. Hakim, T. Spitzer, and J. Armitage, “Sprint: Agile specifications in Shockwave and Flash”, in *Proceedings of the 2003 Conference on Designing for User Experiences - DUX ’03*, ser. DUX ’03, San Francisco, California: ACM Press, 2003, p. 1, ISBN: 978-1-58113-728-6. DOI: 10.1145/997078.997111. [Online]. Available: http://portal.acm.org/citation.cfm?doid=997078.997111 (visited on 01/30/2021).

[203] A. Poth, M. Werner, and X. Lei, “How to Deliver Faster with CI/CD Integrated Testing Services?”, in *European Conference on Software Process Improvement*, X. Larrucea, I. Santamaria, R. V. O’Connor, and R. Messnarz, Eds., Springer, vol. 896, Cham: Springer International Publishing, 2018, pp. 401–409. DOI: 10.1007/978-3-319-97925-0_33. [Online]. Available: http://link.springer.com/10.1007/978-3-319-97925-0_33 (visited on 01/30/2021).

[204] S. Mäkinen, M. Leppänen, T. Kilamo, *et al.*, “Improving the Delivery Cycle: A Multiple-Case Study of the Toolchains in Finnish Software Intensive Enterprises”, *Information and Software Technology*, vol. 80, pp. 175–194, Dec. 2016, ISSN: 09505849. DOI: 10.1016/j.infsof.2016.09.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584916301434 (visited on 01/30/2021).

[205] ISO, *ISO/IEC 25010:2011 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQUARE) — System and Software Quality Models.* Geneva: CH: ISO Geneva, 2011, ISBN: 978 0 580 70223 5. [Online]. Available: https://www.iso.org/standard/35733.html.

[206] D. L. Parnas, “Software Aging”, in *Proceedings of the 16th International Conference on Software Engineering*, T. Richard N. and C. Joëlle, Eds., ser. ICSE ’94, Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 279–287, ISBN: 0-8186-5855-X. DOI: 10.5555/257734.257788.

[207] M. Fowler. “Specification By Example”, SpecificationByExample. (2011), [Online]. Available: https://martinfowler.com/bliki/SpecificationByExample.html.

[208] V. Vernon, *Implementing Domain-Driven Design.* Addison-Wesley, 2013, ISBN: 0-321-83457-7.

[210] S. Brown, *Software Architecture for Developers Technical Leadership by Coding, Coaching, Collaboration, Architecture Sketching and Just Enough up Front Design.* Leanpub, 2014, vol. 7. [Online]. Available: http://leanpub.com/software-architecture-for-developers.

[211] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, 2009, ISBN: 978-0-13-235088-4.

[212] A. Ampatzoglou, S. Bibi, P. Avgeriou, M. Verbeek, and A. Chatzigeorgiou, "Identifying, Categorizing and Mitigating Threats to Validity in Software Engineering Secondary Studies", *Information and Software Technology*, 2018. DOI: 10.1016/j.infsof.2018.10.006. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.10.006.

[213] Y. Beshawred, *Open source & SaaS tools | StackShare*, Jul. 2020. [Online]. Available: https://stackshare.io/categories.

[214] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for Including Grey Literature and Conducting Multivocal Literature Reviews in Software Engineering", *Information and Software Technology*, vol. 106, pp. 101–121, 5 Feb. 2019, ISSN: 09505849. DOI: 10.1016/j.infsof.2018.09.006.

[215] R. Yin, *Case Study Research*, Fourth Edition. Thousand Oaks, CA: Sage Publications, Inc., Dec. 4, 2008, ISBN: 978-1-4129-6099-1.

[216] Open Source Community, *Bminor/Bash: Unofficial Mirror of Bash Repository. Updated Daily.* 2020. [Online]. Available: https://github.com/bminor/bash.

[217] Anonymous. "Firefox Source Code Directory Structure — Firefox Source Docs Documentation". (Jan. 2021), [Online]. Available: https://firefox-source-docs.mozilla.org/contributing/directory_structure.html.

[218] Open Source Community, *Latex2e/Base/Doc at Master ˙ Latex3/Latex2e*, 2020. [Online]. Available: https://github.com/latex3/latex2e/tree/master/base/doc.

[219] M. Fowler, *Technology Radar | an Opinionated Guide to Technology Frontiers | Thoughtworks*, Oct. 2020. [Online]. Available: https://www.thoughtworks.com/radar.

[220] Gartner, *Hype Cycle Research Methodology*, Oct. 2020. [Online]. Available: https://www.gartner.com/en/research/methodologies/gartner-hype-cycle.

[221] NetApplications. "Search Engine Market Share". (Oct. 2020), [Online]. Available: https://netmarketshare.com/search-engine-market-share.aspx?.

[222] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering", *Engineering*, vol. 2, p. 1051, 4ve 2007, ISSN: 00010782. [Online]. Available: http://cdn.elsevier.com/promis_misc/525444systematicreviewsguide.pdf.

[223] C. S. Peirce, N. Houser, and C. J. W. Kloesel, *The Essential Peirce: Selected Philosophical Writings*, in collab. with P. E. Project. Bloomington: Indiana University Press, 1992, ISBN: 978-0-253-21190-3.

[224] M. McLuhan and Q. Fiore, "The Medium Is the Message", *New York Times*, vol. 123, pp. 126–128, 1967. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/08956308.2016.1209068.

[225] D. Kipperman, "Teaching Through Technology Concepts", *Strengthening the position of technology education in the curriculum*, 2009. DOI: 10.1007/978-3-319-38889-2_15-1.

[226] S. Jansen, A. Finkelstein, and S. Brinkkemper, "A Sense of Community: A Research Agenda for Software Ecosystems", *2009 31st International Conference on Software Engineering - Companion Volume*, no. June, pp. 187–190, 2009. DOI: 10.1109/ICSE-COMPANION.2009.5070978. [Online]. Available: http://ieeexplore.ieee.org/document/5070978/ (visited on 01/30/2021).

[227] B. Fitzgerald and K.-J. Stol, "Continuous Software Engineering and Beyond: Trends and Challenges", in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, M. Tichy, J. Bosch, M. Goedicke, and M. Larsson, Eds., ser. RCoSE 2014, New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–9, ISBN: 978-1-4503-2856-2. DOI: 10.1145/2593812.2593813. [Online]. Available: https://doi.org/10.1145/2593812.2593813.

[228] J. Cohen, "Special issue: Digital libraries", *Communications of the ACM*, vol. 39, no. 11, Nov. 1996.

[229] J. 1. Field, *Lifelong Learning and the New Educational Order*. Stoke-on-Trent: Trentham, 2000, ISBN: 978-1-85856-198-1.

[230] L. Uden and A. Dix, "Lifelong Learning for Software Engineers", *International Journal of Continuing Engineering Education and Life Long Learning*, vol. 14, no. 1-2, pp. 101–110, 2004. DOI: 10.1504/IJCEELL.2004.004578.

[231] P. Kruchten, "Lifelong Learning for Lifelong Employment", *IEEE Software*, vol. 32, no. 4, pp. 85–87, 2015. DOI: 10.1109/MS.2015.97.

[232] L. K. Baartman and E. De Bruijn, "Integrating Knowledge, Skills and Attitudes: Conceptualising Learning Processes Towards Vocational Competence", *Educational Research Review*, vol. 6, no. 2, pp. 125–134, 2011, ISSN: 1747938X. DOI: 10.1016/j.edurev.2011.03.001. [Online]. Available: http://dx.doi.org/10.1016/j.edurev.2011.03.001.

[233] D. A. Schön, *The Reflective Practitioner*. New York, NY, USA: Basic Books, Inc., 1984, ISBN: 978-0-203-96337-1. DOI: 10.4324/9780203963371. [Online]. Available: https://doi.org/10.4324/9780203963371.

[234] J. Kruger and D. Dunning, "Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments.", *Journal of personality and social psychology*, vol. 77, no. 6, p. 1121, 1999. DOI: 10.1037//0022-3514.77.6.1121.

[235] M. Csikszentmihalyi, S. Abuhamdeh, and J. Nakamura, *Flow*. London, UK: Springer, 2014, ISBN: 0-06-133920-2.

[236] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Berkeley, CA: Oxford university press, 1977, ISBN: 0-19-501919-9.

[237] J. O. Coplien. "Canonical Form". (2021), [Online]. Available: http://wiki.c2.com/?CanonicalForm (visited on 04/02/2021).

[238] J. Bergin, J. Eckstein, M. Volter, *et al.*, *Pedagogical Patterns: Advice for Educators*. Pleasantville, NY: Joseph Bergin Software Tools, 2012, ISBN: 1-4791-7182-4.

[239] D. Jones. "Software Effort Estimation Is Mostly Fake Research", The Shape of Code. (Jan. 17, 2021), [Online]. Available: http://shape-of-code.coding-guidelines.com/2021/01/17/software-effort-estimation-is-mostly-fake-research/ (visited on 01/30/2021).

[240] T. Kuhn, *The Structure of Scientific Revolutions*. Princeton University Press, 1970, vol. 111, ISBN: 978-0-226-45812-0.

[241] I. Douven, "Abduction", in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Summer 2021, Metaphysics Research Lab, Stanford University, 2021. [Online]. Available: https://plato.stanford.edu/archives/sum2021/entries/abduction/ (visited on 04/25/2022).

[242] G. Stevens, M. Rohde, M. Korn, and V. Wulf, "Grounded Design: A Research Paradigm in Practice-Based Computing", in *Socio-Informatics*, V. Wulf, V. Pipek, D. Randall, M. Rohde, K. Schmidt, and G. Stevens, Eds., Oxford University Press, 2018.

[243] S. T. March and G. F. Smith, "Design and Natural Science Research on Information Technology", *Decision support systems*, vol. 15, no. 4, pp. 251–266, 1995, ISSN: 0167-9236. DOI: 10.1016/0167-9236(94)00041-2.

[244] H. A. Simon, *The Sciences of the Artificial*. Cambridge, Massachusetts: MIT Press, 1996, ISBN: 978-0-262-19374-0. [Online]. Available: https://doi.org/10.7551/mitpress/12107.001.0001.

[245] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research", *MIS quarterly*, vol. 28, pp. 75–105, 2004. DOI: 10.2307/25148625. [Online]. Available: https://www.jstor.org/stable/25148625.

[246] B. Dunbar. "Technology Readiness Level", NASA. (Apr. 2021), [Online]. Available: https://www.nasa.gov/directorates/heo/scan/engineering/technology/technology_readiness_level (visited on 06/17/2021).

[247] G. Sperling, "The Information Available in Brief Visual Presentations.", *Psychological monographs: General and applied*, vol. 74, no. 11, pp. 1–29, 1960. DOI: 10.1037/h0093759.

[248] S. Ainsworth, "DeFT: A Conceptual Framework for Considering Learning With Multiple Representations", *Learning and instruction*, vol. 16, no. 3, pp. 183–198, 2006. DOI: 10.1016/j.learninstruc.2006.03.001.

[249] M. Drury, K. Conboy, and K. Power, "Obstacles to Decision Making in Agile Software Development Teams", *Journal of Systems and Software*, vol. 85, no. 6, pp. 1239–1254, 2012. DOI: 10.1016/j.jss.2012.01.058.

[250] N. B. Moe, A. Aurum, and T. Dybå, "Challenges of Shared Decision-Making: A Multiple Case Study of Agile Software Development", *Information and Software Technology*, vol. 54, no. 8, pp. 853–865, 2012, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2011.11.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584911002308.

[251] W. B. Rouse, "Agile Information Systems for Agile Decision Making", in *Agile Information Systems*, K. C. DeSouza, Ed., London: Routledge, 2007, pp. 16–30, ISBN: 978-0-08-046368-1.

[252] Anonymous. "STEPE—Social, Technical, Economic, Political and Ecological Factor Model". (1990), [Online]. Available: https://pages.gseis.ucla.edu/faculty/richardson/STEPE.htm (visited on 08/27/2022).

[253] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen, "New Directions on Agile Methods: A Comparative Analysis", in *Proceedings 25th International Conference on Software Engineering 2003*, ser. Proceedings - International Conference on Software Engineering, IEEE, vol. 2003, Portland: IEEE Institute of Electrical and Electronic Engineers, 2003, pp. 244–254. DOI: 10.1109/ICSE.2003.1201204.

[254] T. Dybå and T. Dingsøyr, "Empirical Studies of Agile Software Development: A Systematic Review", *Information and Software Technology*, vol. 50, no. 9-10, pp. 833–859, Aug. 2008, ISSN: 09505849. DOI: 10.1016/j.infsof.2008.01.006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584908000256 (visited on 01/30/2021).

[255] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe, "Cost, Benefits and Quality of Software Development Documentation: A Systematic Mapping", *Journal of Systems and Software*, vol. 99, pp. 175–198, Jan. 2015, ISSN: 01641212. DOI: 10.1016/j.jss.2014.09.042. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121214002131 (visited on 01/30/2021).

[256] S. Committee, "ISO/IEC/IEEE International Standard - Systems and Software Engineering – Life Cycle Processes –Requirements Engineering", *ISO/IEC/IEEE 29148:2011(E)*, pp. 1–94, Dec. 2011. DOI: 10.1109/IEEESTD.2011.6146379.

[257] U. Van Heesch, V.-P. Eloranta, P. Avgeriou, K. Koskimies, and N. Harrison, "Decision-Centric Architecture Reviews", *IEEE Software*, vol. 31, no. 1, pp. 69–76, 2014. DOI: 10.1109/MS.2013.22.

[258] E. Ries. "Minimum Viable Product: A Guide". (Aug. 2009), [Online]. Available: http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html (visited on 11/01/2021).

[259] M. A. Cohen, J. Eliasberg, and T.-H. Ho, "New Product Development: The Performance and Time-to-Market Tradeoff", *Management Science*, vol. 42, no. 2, pp. 173–186, 1996, ISSN: 0025-1909. DOI: 10.1287/mnsc.42.2.173. [Online]. Available: http://pubsonline.informs.org/doi/abs/10.1287/mnsc.42.2.173.

[260] K. E. W. Morand, "Software Requirements As Executable Code", Regis University, Dayton Memorial Library, Thesis, 2012. [Online]. Available: https://epublications.regis.edu/theses/232/.

[261] A. Silva, T. Araújo, J. Nunes, *et al.*, "A Systematic Review on the Use of Definition of Done on Agile Software Development Projects", in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 2017, pp. 364–373. DOI: 10.1145/3084226.3084262.

[262] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus, "What Do We Know About Test-Driven Development?", *IEEE software*, vol. 27, no. 6, pp. 16–19, 2010. DOI: 10.1109/MS.2010.152.

[263] M. Ghafari, T. Gross, D. Fucci, and M. Felderer, "Why Research on Test-Driven Development Is Inconclusive?", in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–10. DOI: 10.1145/3382494.3410687.

[264] C. Solis and X. Wang, "A Study of the Characteristics of Behaviour Driven Development", in *37th EUROMICRO Conference on Software Engineering and Advanced Applications*, IEEE, 2011, pp. 383–387. DOI: 10.1109/SEAA.2011.76.

[265] A. Scandaroli, R. Leite, A. H. Kiosia, and S. A. Coelho, "Behavior-Driven Development as an Approach to Improve Software Quality and Communication Across Remote Business Stakeholders, Developers and QA: Two Case Studies", in *Proceedings of the 14th International Conference on Global Software Engineering*, ser. ICGSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 105–110. DOI: 10.1109/ICGSE.2019.00016.

[266] K. Pugh, *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration.* Boston, MA: Pearson Education, 2010, ISBN: 978-0-321-71408-4.

[267] S. Park and F. Maurer, "A Literature Review on Story Test Driven Development", in *Agile Processes in Software Engineering and Extreme Programming*, A. Sillitti, A. Martin, X. Wang, and E. Whitworth, Eds., Springer, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 208–213, ISBN: 978-3-642-13054-0. DOI: 10.1007/978-3-642-13054-0_20.

[268] B. Losada, J.-M. López-Gil, and M. Urretavizcaya, "Improving Agile Software Development Methods by Means of User Objectives: An End User Guided Acceptance Test-Driven Development Proposal", in *Proceedings of the XX International Conference on Human Computer Interaction*, ser. Interacción '19, New York, NY, USA: Association for Computing Machinery, 2019, ISBN: 978-1-4503-7176-6. DOI: 10.1145/3335595.3335650.

[269] F. Beetz and S. Harrer, "Gitops: The Evolution of Devops?", *IEEE Software*, vol. 39, no. 4, pp. 70–75, 2022. DOI: 10.1109/MS.2021.3119106.

[270] F. F.-H. Nah, S. Faja, and T. Cata, "Characteristics of ERP Software Maintenance: A Multiple Case Study", *Journal of software maintenance and evolution: research and practice*, vol. 13, no. 6, pp. 399–414, 2001. DOI: 10.1002/smr.239.

[271] C. Ghezzi, "Of Software and Change", *Journal of Software: Evolution and Process*, vol. 29, no. 9, e1888, 2017. DOI: 10.1002/smr.1888.

[272] R. Cross and L. Sproull, "More Than an Answer: Information Relationships for Actionable Knowledge", *Organization science*, vol. 15, no. 4, pp. 446–462, 2004. DOI: 10.1287/orsc.1040.0075.

[273] J. Lighthill, "Artificial Intelligence: A General Survey. Science Research Council", Science Research Council (SRC), Government Report, 1973. [Online]. Available: http://www.chilton-computing.org.uk/inf/literature/reports/lighthill_report/p001.htm (visited on 12/02/2021).

[274] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach, Global Edition*, 4th ed. London, England: Pearson Education, 2021, ISBN: 978-1-292-40117-1.

[275] D. H. Hubel and T. N. Wiesel, "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex", *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154, 1962. DOI: 10.1113/jphysiol.1962.sp006837. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/14449617/.

[276] J. L. McClelland, D. E. Rumelhart, P. R. Group, *et al.*, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations.* MIT press Cambridge, MA, 1986, vol. 1, ISBN: 978-0-262-29140-8. [Online]. Available: https://doi.org/10.7551/mitpress/5236.001.0001.

[277] F. V. Veen. "The Neural Network Zoo", The Asimov Institute. (Sep. 2016), [Online]. Available: https://www.asimovinstitute.org/neural-network-zoo/ (visited on 08/27/2022).

[278] Anonymous. "Conventional Commits", Conventional Commits. (2022), [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/ (visited on 08/27/2022).

[279] L. DalleMule and T. H. Davenport, "What's Your Data Strategy", *Harvard Business Review*, vol. 95, no. 3, pp. 112–121, 2017. [Online]. Available: https://hbr.org/webinar/2017/04/whats-your-data-strategy.

[280] J. Yang, S. C. Han, and J. Poon, "A Survey on Extraction of Causal Relations From Natural Language Text", *Knowledge and Information Systems*, vol. 64, no. 5, pp. 1161–1186, 2022, ISSN: 01641212. DOI: 10.1007/s10115-022-01665-w.

[281] T. O. Motta, R. R. Gomes e Souza, and C. Sant'Anna, "Characterizing Architectural Information in Commit Messages: An Exploratory Study", in *Proceedings of the XXXII Brazilian Symposium on Software Engineering - SBES '18*, U. Kulesza, R. Prikladnicki, M. A. Gerosa, C. Werner, and R. Andrade, Eds., Sao Carlos, Brazil: ACM Press, 2018, pp. 12–21. DOI: 10.1145/3266237.3266260.

[282] M. F. Porter, "An Algorithm for Suffix Stripping", *Program: electronic library and information systems*, vol. 14, no. 3, pp. 130–137, 1980, ISSN: 0033-0337. DOI: 10.1108/eb046814.

[283] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016, ISBN: 978-0-262-03561-3. [Online]. Available: http://www.deeplearningbook.org.

[284] S. Bozinovski, "Reminder of the First Paper on Transfer Learning in Neural Networks, 1976", *Informatica*, vol. 44, no. 3, 2020. DOI: 10.31449/inf.v44i3.2828.

[285] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018. DOI: 10.48550/arXiv.1810.04805.

[286] R. K. Yin, *Qualitative Research from Start to Finish*, Second edition. New York London: The Guilford Press, 2016, ISBN: 978-1-4625-1797-8.

[287] A. S. Lee and G. S. Hubona, "A Scientific Basis for Rigor in Information Systems Research", *MIS quarterly*, pp. 237–262, 2009. DOI: 10.2307/20650291.

[288] R. A. Gonzalez and H. G. Sol, "Validation and Design Science Research in Information Systems", in *Research Methodologies, Innovations and Philosophies in Software Systems Engineering and Information Systems*, M. Mora, Ed., IGI Global, 2012, pp. 403–426, ISBN: 978-1-4666-0179-6.

[289] C. Marshall and G. B. Rossman, *Designing Qualitative Research*. London: SAGE Publications, Inc., Jan. 7, 2015, ISBN: 978-1-4522-7100-2.

[290] Standards Committee, *IEEE Recommended Practice for Software Requirements Specifications*. 1998, vol. 1998, p. 37, ISBN: 0-7381-0332-2. [Online]. Available: http://www.math.uaa.alaska.edu/%7B%%7D7B%7B~%7D%7B%%7D7Dafkjm/cs401/IEEE830.pdf.

[291] Bureau of Labor Statistics, U.S. Department of Labor. "Average Number of Years That Employees Stayed In The Top 10 Biggest Companies in Tech", MobileMonkey. (2022), [Online]. Available: https://mobilemonkey.com/articles/employee-tenure-in-tech-companies/ (visited on 09/11/2022).

[292] Anonymous. "HackerLife", hackerlife. (2017), [Online]. Available: https://hackerlife.co (visited on 09/11/2022).

[293] W. Hasselbring, L. Carr, S. Hettrick, H. Packer, and T. Tiropanis, "Open Source Research Software", *Computer*, vol. 53, no. 8, pp. 84–88, Jul. 2020. DOI: 10.1109/MC.2020.2998235.

[294] K. R. Larsen, R. Lukyanenko, R. M. Mueller, *et al.*, "Validity in Design Science Research", in *International Conference on Design Science Research in Information Systems and Technology*, S. Hoffman, O. Müller, and M. Rossi, Eds., ser. Lecture Notes in Computer Science, Springer, vol. 12388, Springer, 2020, pp. 272–282, ISBN: 978-3-030-64823-7. DOI: 10.1007/978-3-030-64823-7_25.

[295] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting After the Fact: Recovering Architectural Design Decisions", *The Journal of Systems & Software*, vol. 81, no. 4, pp. 536–557, 2008. DOI: 10.1016/j.jss.2007.08.025. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S016412120700194X.

[296] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic, "Recovering Architectural Design Decisions", in *2018 IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA: IEEE, Apr. 2018, pp. 95–9509, ISBN: 978-1-5386-6398-1. DOI: 10.1109/ICSA.2018.00019. [Online]. Available: https://ieeexplore.ieee.org/document/8417122/ (visited on 11/26/2022).

[297] H. Touvron, T. Lavril, G. Izacard, *et al.* "LLaMA: Open and Efficient Foundation Language Models". (Feb. 27, 2023), [Online]. Available: http://arxiv.org/abs/2302.13971 (visited on 05/15/2023), preprint.

List of References in Mapping Study

[S31] T. Theunissen and U. Van Heesch, "Specification in Continuous Software Development", in *Proceedings of the 22ND European Conference on Pattern Languages of Programs*, A. for Computing Machinery, Ed., ser. EuroPLoP '17, ACM, New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–19, ISBN: 978-1-4503-4848-5. DOI: 10.1145/3147704.3147709.

[S32] U. Van Heesch, T. Theunissen, O. Zimmermann, and U. Zdun, "Software Specification and Documentation in Continuous Software Development: A Focus Group Report", in *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, (Irsee, Germany), C. Preschern and C. Kreiner, Eds., ser. EuroPLoP '17, New York, NY, USA: ACM, 2017, 35:1–35:13, ISBN: 978-1-4503-4848-5. DOI: 10.1145/3147704.3147742. [Online]. Available: http://doi.acm.org/10.1145/3147704.3147742.

[S60] B. K. Beck and P. Date, *Test-Driven Development by Example.* 2002, ISBN: 0-321-14653-0.

[S61] U. van Heesch, P. Avgeriou, and R. Hilliard, "A Documentation Framework for Architecture Decisions", *Journal of Systems and Software*, vol. 85, no. 4, pp. 795–820, Apr. 2012, ISSN: 01641212. DOI: 10.1016/j.jss.2011.10.017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121211002755 (visited on 01/30/2021).

[S74] G. Borrego, A. L. Morán, R. R. Palacio Cinco, O. M. Rodríguez-Elias, and E. García-Canseco, "Review of Approaches to Manage Architectural Knowledge in Agile Global Software Development", *IET Software*, vol. 11, no. 3, pp. 77–88, Jun. 2017. DOI: 10.1049/iet-sen.2016.0197. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1049/iet-sen.2016.0197 (visited on 01/30/2021).

[S75] J. Nawrocki, M. Jasinski, B. Walter, and A. Wojciechowski, "Extreme Programming Modified: Embrace Requirements Engineering Practices", in *Proceedings IEEE Joint International Conference on Requirements Engineering*, Essen, Germany: IEEE Comput. Soc, 2002, pp. 303–310, ISBN: 978-0-7695-1465-9. DOI: 10.1109/ICRE.2002.1048543. [Online]. Available: http://ieeexplore.ieee.org/document/1048543/ (visited on 01/30/2021).

[S76] C. J. Stettina and E. Kroon, "Is There an Agile Handover? An Empirical Study of Documentation and Project Handover Practices Across Agile Software Teams", in *2013 International Conference on Engineering, Technology and Innovation, ICE 2013 and IEEE International Technology Management Conference, ITMC 2013*, The Hague, Netherlands: IEEE, 2015, pp. 1–12, ISBN: 978-1-4673-7383-8. DOI: 10.1109/ITMC.2013.7352703. [Online]. Available: https://www.dropbox.com/home/publications?preview=stettina2013.pdf.

[S86] B. Fitzgerald and K.-J. Stol, "Continuous Software Engineering: A Roadmap and Agenda", *Journal of Systems and Software*, vol. 123, pp. 176–189, Jan. 2017, ISSN: 01641212. DOI: 10.1016/j.jss.2015.06.063. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121215001430 (visited on 01/30/2021).

[S89] C. Manteuffel, D. Tofan, H. Koziolek, T. Goldschmidt, and P. Avgeriou, "Industrial Implementation of a Documentation Framework for Architectural Decisions", in *2014 IEEE/IFIP Conference on Software Architecture*, IEEE, Sydney, Australia: IEEE, Apr. 2014, pp. 225–234, ISBN: 978-1-4799-3412-6. DOI: 10.1109/WICSA.2014.32. [Online]. Available: http://ieeexplore.ieee.org/document/6827122/ (visited on 01/30/2021).

[S97] M. Kersten, "A Cambrian Explosion of DevOps Tools", *IEEE Softw.*, vol. 35, no. 2, pp. 14–17, Mar. 2018, ISSN: 0740-7459. DOI: 10.1109/MS.2018.1661330. [Online]. Available: http://ieeexplore.ieee.org/document/8314153/ (visited on 01/30/2021).

[S111] S. Voigt, J. von Garrel, J. Müller, and D. Wirth, "A Study of Documentation in Agile Software Projects", in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Ciudad Real Spain: ACM, Sep. 8, 2016, pp. 1–6, ISBN: 978-1-4503-4427-2. DOI: 10.1145/2961111.2962616. [Online]. Available: https://dl.acm.org/doi/10.1145/2961111.2962616 (visited on 01/30/2021).

[S112] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A Study of the Documentation Essential to Software Maintenance", in *Proceedings of the 23rd Annual International Conference on Design of Communication Documenting & Designing for Pervasive Information - SIGDOC '05*, Coventry, United Kingdom: ACM Press, 2005, p. 68, ISBN: 978-1-59593-175-7. DOI: 10.1145/1085313.1085331. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1085313.1085331 (visited on 01/30/2021).

[S114] S. Voigt, D. Huttemann, and A. Gohr, "SprintDoc: Concept for an Agile Documentation Tool", in *2016 11th Iberian Conference on Information Systems and Technologies (CISTI)*, Gran Canaria, Spain: IEEE, Jun. 2016, pp. 1–6, ISBN: 978-989-98434-6-2. DOI: 10.1109/CISTI.2016.7521550. [Online]. Available: http://ieeexplore.ieee.org/document/7521550/ (visited on 01/30/2021).

[S115] E. Rubin and H. Rubin, "Supporting Agile Software Development Through Active Documentation", *Requirements Eng*, vol. 16, no. 2, pp. 117–132, Jun. 2011. DOI: 10.1007/s00766-010-0113-9. [Online]. Available: http://link.springer.com/10.1007/s00766-010-0113-9 (visited on 01/30/2021).

[S116] S. Gerdes, S. Jasser, M. Riebisch, S. Schröder, M. Soliman, and T. Stehle, "Towards the Essentials of Architecture Documentation for Avoiding Architecture Erosion", in *Proccedings of the 10th European Conference on Software Architecture Workshops - ECSAW '16*, ser. ECSAW '16, New York, NY, USA: ACM, 2016, pp. 1–4, ISBN: 978-1-4503-4781-5. DOI: 10.1145/2993412.3004844. [Online]. Available: http://doi.acm.org/10.1145/2993412.3004844%20https://www.dropbox.com/home/publications?preview=gerdes2016.pdf%20http://dl.acm.org/citation.cfm?doid=2993412.3004844.

[S117] T. Sauer, "Using Design Rationales for Agile Documentation", in *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.*, Linz, Austria: IEEE Comput. Soc, 2003, pp. 326–331, ISBN: 978-0-7695-1963-0. DOI: 10.1109/ENABL.2003.1231431. [Online]. Available: http://ieeexplore.ieee.org/document/1231431/ (visited on 01/30/2021).

[S118] R. Ankori, "Automatic Requirements Elicitation in Agile Processes", in *Proceedings - IEEE International Conference on Software - Science, Technology and Engineering 2005, SwSTE '05*, vol. 2005, 2005, pp. 101–109, ISBN: 0-7695-2335-8. DOI: 10.1109/SWSTE.2005.8. [Online]. Available: https://www.dropbox.com/home/publications?preview=10.1109%7B%%7D40SWSTE.2005.8.pdf.

[S119] A. Bollin and D. Rauner-Reithmayer, "Formal Specification Comprehension: The Art of Reading and Writing Z", in *Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering - FormaliSE 2014*, Hyderabad, India: ACM Press, 2014, pp. 3–9, ISBN: 978-1-4503-2853-1. DOI: 10.1145/2593489.2593491. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2593489.2593491 (visited on 01/30/2021).

[S120] G. Borrego, "Condensing Architectural Knowledge from Unstructured Textual Media in Agile GSD Teams", in *2016 IEEE 11th International Conference on Global Software Engineering Workshops (ICGSEW)*, Orange County, CA, USA: IEEE, Aug. 2016, pp. 69–72, ISBN: 978-1-5090-3625-7. DOI: 10.1109/ICGSEW.2016.16. [Online]. Available: http://ieeexplore.ieee.org/document/7579489/ (visited on 01/30/2021).

[S121] G. Borrego, A. L. Moran, and R. Palacio, "Preliminary Evaluation of a Tag-Based Knowledge Condensation Tool in Agile and Distributed Teams", in *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*, Buenos Aires, Argentina: IEEE, May 2017, pp. 51–55, ISBN: 978-1-5386-1587-4. DOI: 10.1109/ICGSE.2017.14. [Online]. Available: http://ieeexplore.ieee.org/document/7976687/ (visited on 01/30/2021).

[S122] L. Briand, "Software Documentation: How Much Is Enough?", in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, Benevento, Italy: IEEE Comput. Soc, 2003, pp. 13–15, ISBN: 978-0-7695-1902-9. DOI: 10.1109/CSMR.2003.1192406. [Online]. Available: http://ieeexplore.ieee.org/document/1192406/ (visited on 01/30/2021).

[S123] J. Choudhury and B. Thushara, “Software Documentation in a Globally Distributed Environment”, in *2014 IEEE 9th International Conference on Global Software Engineering*, Shanghai, China: IEEE, Aug. 2014, pp. 90–94, ISBN: 978-1-4799-4360-9. DOI: 10.1109/ICGSE.2014.23. [Online]. Available: http://ieeexplore.ieee.org/document/6915258/ (visited on 01/30/2021).

[S124] H. B. Christensen and K. M. Hansen, “Towards Architectural Information in Implementation”, in *Proceeding of the 33rd International Conference on Software Engineering - ICSE ’11*, 2011, p. 928, ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985948.

[S125] E. Di Nitto, P. Jamshidi, M. Guerriero, I. Spais, and D. A. Tamburri, “A Software Architecture Framework for Quality-Aware Devops”, in *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, Saarbrücken Germany: ACM, Jul. 21, 2016, pp. 12–17, ISBN: 978-1-4503-4411-1. DOI: 10.1145/2945408.2945411. [Online]. Available: https://dl.acm.org/doi/10.1145/2945408.2945411 (visited on 01/30/2021).

[S126] Prashant Gandhi, N. Haugen, M. Hill, and R. Watt, “Creating a Living Specification Using FIT Documents”, in *Agile Development Conference (ADC’05)*, Denver, CO, USA: IEEE Comput. Soc, 2005, pp. 253–258, ISBN: 978-0-7695-2487-0. DOI: 10.1109/ADC.2005.19. [Online]. Available: http://ieeexplore.ieee.org/document/1609829/ (visited on 01/30/2021).

[S127] I. Hadar, S. Sherman, E. Hadar, and J. J. Harrison, “Less Is More: Architecture Documentation for Agile Development”, in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, San Francisco, CA, USA: IEEE, May 2013, pp. 121–124, ISBN: 978-1-4673-6290-0. DOI: 10.1109/CHASE.2013.6614746. [Online]. Available: http://ieeexplore.ieee.org/document/6614746/ (visited on 01/30/2021).

[S128] A. Hess, P. DIebold, and N. Seyff, “Towards Requirements Communication and Documentation Guidelines for Agile Teams”, in *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference Workshops, REW 2017*, Lisbon, Portugal: IEEE, 2017, pp. 415–418, ISBN: 978-1-5386-3488-2. DOI: 10.1109/REW.2017.64. [Online]. Available: https://www.dropbox.com/home/publications?preview=hess2017.pdf.

[S129] A. Jarzębowicz and K. Połocka, “Selecting Requirements Documentation Techniques for Software Projects: A Survey Study”, in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 24, 2017, pp. 1189–1198, ISBN: 978-83-946253-7-5. DOI: 10.15439/2017F387. [Online]. Available: https://fedcsis.org/proceedings/2017/drp/387.html (visited on 01/30/2021).

[S130] R. K. Kavitha and M. S. Irfan Ahmed, “A Knowledge Management Framework for Agile Software Development Teams”, in *2011 International Conference on Process Automation, Control and Computing*, Coimbatore, Tamilnadu, India: IEEE, Jul. 2011, pp. 1–5, ISBN: 978-1-61284-765-8. DOI: 10.1109/PACC.2011.5978877. [Online]. Available: http://ieeexplore.ieee.org/document/5978877/ (visited on 01/30/2021).

[S131] A. I. M. Leite, “An Approach to Support the Specification of Agile Artifacts in the Development of Safety-Critical Systems”, in *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference, RE 2017*, Lisbon, Portugal: IEEE, 2017, pp. 526–531, ISBN: 978-1-5386-3191-1. DOI: 10.1109/RE.2017.43. [Online]. Available: https://www.dropbox.com/home/publications?preview=leite2017.pdf.

[S133] J. Medeiros, A. Vasconcelos, M. Goulão, C. Silva, and J. Araújo, “An Approach Based on Design Practices to Specify Requirements in Agile Projects”, in *Proceedings of the Symposium on Applied Computing - SAC ’17*, ser. SAC ’17, New York, NY, USA: ACM, 2017, pp. 1114–1121, ISBN: 978-1-4503-4486-9. DOI: 10.1145/3019612.3019753. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019753%20https://www.dropbox.com/home/publications?preview=p1114-medeiros.pdf%20http://dl.acm.org/citation.cfm?doid=3019612.3019753.

[S134] S. Pinna, P. Lorrai, M. Marchesi, and N. Serra, “Developing a Tool Supporting XP Process”, in *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, F. Maurer and D. Wells, Eds., ser. LECTURE NOTES IN COMPUTER SCIENCE, vol. 2753, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 151–160. DOI: 10.1007/978-3-540-45122-8_17. [Online]. Available: http://link.springer.com/10.1007/978-3-540-45122-8_17 (visited on 01/30/2021).

[S135] C. R. Prause and Z. Durdik, “Architectural Design and Documentation: Waste in Agile Development?”, in *2012 International Conference on Software and System Process, ICSSP 2012 - Proceedings*, ser. ICSSP ’12, Piscataway, NJ, USA: IEEE Press, Jun. 2012, pp. 130–134, ISBN: 978-1-4673-2352-9. DOI: 10.1109/ICSSP.2012.6225956. [Online]. Available: https://www.dropbox.com/home/publications?preview=prause2012.pdf%20http://dl.acm.org/citation.cfm?id=2664360.2664380%20https://www.dropbox.com/home/publications?preview=10.0000%7B%%7D40dl.acm.org%7B%%7D402664360.2664380.pdf%20http://dx.doi.org/10.1109/ICSSP.2012.6225956.

[S136] S. Saito, Y. Iimura, A. K. Massey, and A. I. Anton, “How Much Undocumented Knowledge Is There in Agile Software Development?: Case Study on Industrial Project Using Issue Tracking System and Version Control System”, in *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference, RE 2017*, Lisbon, Portugal: IEEE, 2017, pp. 194–203, ISBN: 978-1-5386-3191-1. DOI: 10.1109/RE.2017.33. [Online]. Available: https://www.dropbox.com/home/publications?preview=saito2017.pdf.

[S137] H. F. Soares, N. S. R. Alves, T. S. Mendes, M. Mendonca, and R. O. Spinola, “Investigating the Link Between User Stories and Documentation Debt on Software Projects”, in *Proceedings - 12th International Conference on Information Technology: New Generations, ITNG 2015*, Las Vegas, NV, USA: IEEE, 2015, pp. 385–390, ISBN: 978-1-4799-8827-3. DOI: 10.1109/ITNG.2015.68. [Online]. Available: https://www.dropbox.com/home/publications?preview=soares2015.pdf.

[S138] C. J. Stettina and W. Heijstek, “Necessary and Neglected? An Empirical Study of Internaldocumentation in Agile Software Development Teams”, in *Proceedings of the 29th ACM International Conference on Design of Communication - SIGDOC ’11*, ser. SIGDOC ’11, New York, NY, USA: ACM, 2011, p. 159, ISBN: 978-1-4503-0936-3. DOI: 10.1145/2038476.2038509. [Online]. Available: http://doi.acm.org/10.1145/2038476.2038509%20http://dl.acm.org/citation.cfm?doid=2038476.2038509%20https://www.dropbox.com/home/publications?preview=stettina2011.pdf.

[S139] C. J. Stettina, W. Heijstek, and T. E. Fægri, “Documentation Work in Agile Teams: The Role of Documentation Formalism in Achieving a Sustainable Practice”, in *Proceedings - 2012 Agile Conference, Agile 2012*, Dallas, TX, USA: IEEE, 2012, pp. 31–40, ISBN: 978-0-7695-4804-3. DOI: 10.1109/Agile.2012.7. [Online]. Available: https://www.dropbox.com/home/publications?preview=stettina2012.pdf.

[S140] T. Waits and J. Yankel, “Continuous System and User Documentation Integration”, in *2014 IEEE International Professional Communication Conference (IPCC)*, Pittsburgh, PA, USA: IEEE, Oct. 2014, pp. 1–5, ISBN: 978-1-4799-3749-3. DOI: 10.1109/IPCC.2014.7020385. [Online]. Available: http://ieeexplore.ieee.org/document/7020385/ (visited on 01/30/2021).

[S141] J. A. Diaz-Pace, M. Nicoletti, S. Schiaffino, and S. Vidal, “Producing Just Enough Documentation: The Next SAD Version Problem”, in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, C. Le Goues and S. Yoo, Eds., vol. 8636, Cham: Springer International Publishing, 2014, pp. 46–60. DOI: 10.1007/978-3-319-09940-8_4. [Online]. Available: http://link.springer.com/10.1007/978-3-319-09940-8_4 (visited on 01/30/2021).

[S142] L. T. Heeager, “How Can Agile and Documentation-Driven Methods be Meshed in Practice?”, in *Agile Processes in Software Engineering and Extreme Programming*, G. Cantone and M. Marchesi, Eds., vol. 179, Cham: Springer International Publishing, 2014, pp. 62–77. DOI: 10.1007/978-3-319-06862-6. [Online]. Available: http://link.springer.com/10.1007/978-3-319-06862-6_5 (visited on 01/30/2021).

[S143] R. Hoda, J. Noble, and S. Marshall, "How Much Is Just Enough?: Some Documentation Patterns on Agile Projects", in *Proceedings of the 15th European Conference on Pattern Languages of Programs*, ACM, ACM, 2010, 13:1–13:13, ISBN: 978-1-4503-0259-3. DOI: 10.1145/2328909.2328926. [Online]. Available: http://doi.acm.org/10.1145/2328909.2328926.

[S144] ISO, IEC, and IEEE, "Systems and Software Engineering – Developing User Documentation in an Agile Environment", ISO/IEC/IEEE, Technical Report, 2012, pp. 1–36. [Online]. Available: https://doi.org/10.1109/IEEESTD.2012.6170923.

[S145] G. Wagenaar, S. Overbeek, G. Lucassen, S. Brinkkemper, and K. Schneider, "Working Software Over Comprehensive Documentation – Rationales of Agile Teams for Artefacts Usage", *J Softw Eng Res Dev*, vol. 6, no. 1, p. 7, Dec. 2018, ISSN: 2195-1721. DOI: 10.1186/s40411-018-0051-7. [Online]. Available: https://jserd.springeropen.com/articles/10.1186/s40411-018-0051-7 (visited on 01/30/2021).

[S146] A. Aguiar, "Tutorial on Agile Documentation with Wikis", in *Proceedings of the 5th International Symposium on Wikis and Open Collaboration - WikiSym '09*, ser. WikiSym '09, New York, NY, USA: ACM, 2009, p. 1, ISBN: 978-1-60558-730-1. DOI: 10.1145/1641309.1641365. [Online]. Available: http://doi.acm.org/10.1145/1641309.1641365%20http://portal.acm.org/citation.cfm?doid=1641309.1641365.

[S148] H. Aman and R. Ibrahim, "XML-DocTracker: Generating software requirements specification (SRS) from XML schema", in *ICISS 2016 - 2016 International Conference on Information Science and Security*, 2017, pp. 1–5, ISBN: 978-1-5090-5493-0. DOI: 10.1109/ICISSEC.2016.7885872. [Online]. Available: https://www.dropbox.com/home/publications?preview=aman2016.pdf.

[S149] C. O. De Melo, D. S. Cruzes, F. Kon, and R. Conradi, "Interpretative Case Studies on Agile Team Productivity and Management", in *Information and Software Technology*, vol. 55, Feb. 2013, pp. 412–427. DOI: 10.1016/j.infsof.2012.09.004. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584912001875.

[S150] L. Delgadillo and O. Gotel, "Story-Wall: A Concept for Lightweight Requirements Management", in *15th IEEE International Requirements Engineering Conference (RE 2007)*, Delhi, India: IEEE, Oct. 2007, pp. 377–378, ISBN: 978-0-7695-2935-6. DOI: 10.1109/RE.2007.41. [Online]. Available: http://ieeexplore.ieee.org/document/4384210/ (visited on 01/30/2021).

[S151] J. Kaariainen, J. Koskela, P. Abrahamsson, and J. Takalo, "Improving Requirements Management in Extreme Programming with Tool Support - an Improvement Attempt That Failed", in *Proceedings. 30th Euromicro Conference, 2004.*, ser. EUROMICRO '04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 342–351, ISBN: 0-7695-2199-1. DOI: 10.1109/EURMIC.2004.1333389. [Online]. Available: http://ieeexplore.ieee.org/document/1333389/%20https://www.dropbox.com/home/publications?preview=kaariainen2004.pdf.

[S152] K. Könnölä, S. Suomi, T. Mäkilä, T. Jokela, V. Rantala, and T. Lehtonen, "Agile Methods in Embedded System Development: Multiple-Case Study of Three Industrial Cases", *Journal of Systems and Software*, vol. 118, pp. 134–150, Aug. 2016, ISSN: 01641212. DOI: 10.1016/j.jss.2016.05.001. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0164121216300413.

[S153] M. Mahoney, "Telling Stories about Software Evolution", in *2011 AGILE Conference*, Salt Lake City, UT, USA: IEEE, Aug. 2011, pp. 127–130, ISBN: 978-1-61284-426-8. DOI: 10.1109/AGILE.2011.22. [Online]. Available: http://ieeexplore.ieee.org/document/6005493/ (visited on 01/30/2021).

[S154] X. Wang, K. Conboy, and O. Cawley, ""Leagile" Software Development: An Experience Report Analysis of the Application of Lean Approaches in Agile Software Development", *Journal of Systems and Software*, vol. 85, no. 6, pp. 1287–1299, Jun. 2012, ISSN: 01641212. DOI: 10.1016/j.jss.2012.01.061. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121212000404 (visited on 01/30/2021).

[S155] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining Devops Automation for Cloud Applications Using Tosca as Standardized Metamodel", *Future Generation Computer Systems*, vol. 56, pp. 317–332, Mar. 2016, ISSN: 0167739X. DOI: 10.1016/j.future.2015.07.017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X15002496 (visited on 01/30/2021).

[S156] V.-P. Eloranta and K. Koskimies, "Lightweight Architecture Knowledge Management for Agile Software Development", in *Agile Software Architecture*, H. Duran-Limon, C. Y. Laporte, O. Karam, M. Mora, and A. Mishra, Eds., Elsevier, 2014, pp. 189–213, ISBN: 978-0-12-407772-0. DOI: 10.1016/B978-0-12-407772-0.00007-1. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780124077720000071 (visited on 01/30/2021).

[S157] P. O. Antonino, T. Keuler, N. Germann, and B. Cronauer, "A Non-Invasive Approach to Trace Architecture Design, Requirements Specification and Agile Artifacts", in *Proceedings of the 2014 23rd Australian Software Engineering Conference*, ser. ASWEC '14, Washington, DC, USA: IEEE Computer Society, 2014, pp. 220–229, ISBN: 978-1-4799-3149-1. DOI: 10.1109/ASWEC.2014.30. [Online]. Available: https://doi.org/10.1109/ASWEC.2014.30.

[S158] N. Kerzazi and B. Adams, "Who Needs Release and Devops Engineers, and Why?", in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, (Austin, Texas), ser. CSED '16, New York, NY, USA: ACM, 2016, pp. 77–83, ISBN: 978-1-4503-4157-8. DOI: 10.1145/2896941.2896957. [Online]. Available: http://doi.acm.org/10.1145/2896941.2896957.

[S159] S. G. Rojas and J. M. C. Mora, "Source Code Documentation Simul Loco", *Iberian Conference on Information Systems and Technologies, CISTI*, pp. 669–673, 2012, ISSN: 21660727. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84869003196%7B&%7DpartnerID=tZOtx3y1.

[S160] S. S. S. I. Perera, "Continuous Scrum: A Framework to Enhance Scrum with DevOps", in *2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer)*, IEEE, Colombo: IEEE, Sep. 2017, pp. 1–7. DOI: 10.1109/ICTER.2017.8257808. [Online]. Available: http://ieeexplore.ieee.org/document/8257808/.

[S161] S. Jones, J. Noppen, and F. Lettice, "Management Challenges for DevOps Adoption within UK SMEs", in *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, ACM, 2016, pp. 7–11. DOI: 10.1145/2945408.2945410.

[S163] D. Wells. "Extreme Programming: A Gentle Introduction.", Extreme Programming: A Gentle Introduction. (Oct. 8, 2013), [Online]. Available: http://www.extremeprogramming.org/ (visited on 07/26/2020).

[S164] S. R. Palmer and M. Felsing, *A Practical Guide to Feature-Driven Development.* Pearson Education, 2001, ISBN: 0-13-067615-2.

[S165] A. Cockburn, "Crystal Clear [electronic Resource]: A Human-Powered Methodology for Small Teams", in *The Agile Software Development Series*, D. O'Hagan, Ed., Pearson Education, 2005, 1 online resource (xxii, 312 p.) ISBN: 0-201-69947-8. [Online]. Available: http://proquest.safaribooksonline.com/0201699478.

[S166] J. A. Highsmith, "Adaptive Software Development: A Collaborative Approach to Managing Complex Systems", in *Journal of Evolutionary Biology*, J. A. Highsmith, Ed., vol. 12, Addison-Wesley, 2000, p. 392, ISBN: 0-932633-40-4. [Online]. Available: http://books.google.com/books?id=R1ZyQgAACAAJ.

[S167] T. Karvonen, T. Suomalainen, M. Juntunen, T. Sauvola, P. Kuvaja, and M. Oivo, "The CRUSOE Framework: A Holistic Approach to Analysing Prerequisites for Continuous Software Engineering", in *International Conference on Product-Focused Software Process Improvement*, Springer, 2016, pp. 643–661. DOI: 10.1007/978-3-319-49094-6_52.

[S168] J. V. Alavandhar and O. \ cNikiforova, "Several Ideas on Integration of Scrum Practices Within Microsoft Solutions Framework", *Applied Computer Systems*, vol. 21, no. 1, pp. 71–79, 2017. DOI: 10.1515/acss-2017-0010.

[S171] R. P. Maranzato, M. Neubert, and P. Herculano, "Scaling Scrum Step by Step: "The Mega Framework"", in *Proceedings of the 2012 Agile Conference*, ser. AGILE '12, IEEE Computer Society, Washington, DC, USA: IEEE Computer Society, 2012, pp. 79–85, ISBN: 978-0-7695-4804-3. DOI: 10.1109/Agile.2012.22. [Online]. Available: https://doi.org/10.1109/Agile.2012.22.

[S172] R. L. Nord, I. Ozkaya, and P. Kruchten, "Agile in Distress: Architecture to the Rescue", in *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation*, T. Dingsøyr, N. B. Moe, R. Tonelli, S. Counsell, C. Gencel, and K. Petersen, Eds., vol. 199, Cham, 2014, pp. 43–57, ISBN: 978-3-319-14358-3. DOI: 10.1007/978-3-319-14358-3_5. [Online]. Available: http://link.springer.com/10.1007/978-3-319-14358-3_5 (visited on 01/30/2021).

[S173] U. van Heesch and P. Avgeriou, "A Pattern Driven Approach Against Architectural Knowledge Vaporization", in *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee*, Conference Proceedings, 2009, pp. 1–12. [Online]. Available: http://www.cs.rug.nl/%20paris/papers/EPLOP09.pdf.

[S176] W. Aslam and F. Ijaz, "A Quantitative Framework for Task Allocation in Distributed Agile Software Development", *IEEE Access*, vol. 6, pp. 15 380–15 390, 2018. DOI: 10.1109/ACCESS.2018.2803685.

[S177] M. Callanan and A. Spillane, "DevOps: Making It Easy to Do the Right Thing", *IEEE Softw.*, vol. 33, no. 3, pp. 53–59, May 2016. DOI: 10.1109/MS.2016.66. [Online]. Available: https://ieeexplore.ieee.org/document/7436644/ (visited on 01/30/2021).

[S178] P. Kruchten, "The 4+1 View Model of Architecture", *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995, ISSN: 07407459. DOI: 10.1109/52.469759. [Online]. Available: http://ieeexplore.ieee.org/document/469759/ (visited on 01/30/2021).

[S179] D. Ståhl and J. Bosch, "Cinders: The Continuous Integration and Delivery Architecture Framework", *Information and Software Technology*, vol. 83, pp. 76–93, Mar. 2017, ISSN: 09505849. DOI: 10.1016/j.infsof.2016.11.006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S095058491630369X (visited on 01/30/2021).

[S180] F. Cannizzo, G. Marcionetti, and P. Moser, "Evolution of the Tools and Practices of a Large Distributed Agile Team", in *Agile 2008 Conference*, IEEE Computer Society, Toronto, ON, Canada: IEEE, 2008, pp. 513–518, ISBN: 978-0-7695-3321-6. DOI: 10.1109/Agile.2008.32. [Online]. Available: http://ieeexplore.ieee.org/document/4599531/ (visited on 01/30/2021).

[S181] T. Buchmann, "Towards Tool Support for Agile Modeling", in *Proceedings of the 2012 Extreme Modeling Workshop on - XM '12*, ACM, ACM, 2012, pp. 9–14, ISBN: 978-1-4503-1804-4. DOI: 10.1145/2467307.2467310. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2467307.2467310.

[S182] G. Wedemann, "Scrum as a Method of Teaching Software Architecture", in *Proceedings of the 3rd European Conference of Software Engineering Education*, ser. ECSEE'18, New York, NY, USA: ACM, 2018, pp. 108–112, ISBN: 978-1-4503-6383-9. DOI: 10.1145/3209087.3209096. [Online]. Available: http://doi.acm.org/10.1145/3209087.3209096.

[S183] H.-M. Chen, R. Kazman, and S. Haziyev, "Agile Big Data Analytics Development: An Architecture-Centric Approach", in *Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS)*, ser. HICSS '16, Washington, DC, USA: IEEE Computer Society, 2016, pp. 5378–5387, ISBN: 978-0-7695-5670-3. DOI: 10.1109/HICSS.2016.665. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2016.665.

[S184] C. Miyachi, "Agile Software Architecture", *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 2, pp. 1–3, Mar. 2011, ISSN: 01635948. DOI: 10.1145/1943371.1943388. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1943371.1943388.

[S185] L. J. Waguespack and W. T. Schiano, "Scrum Project Architecture and Thriving Systems Theory", in *Proceedings of the 2012 45th Hawaii International Conference on System Sciences*, ser. HICSS '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 4943–4951, ISBN: 978-0-7695-4525-7. DOI: 10.1109/HICSS.2012.513. [Online]. Available: https://doi.org/10.1109/HICSS.2012.513.

[S186] S. Akman, E. B. Aksuyek, and O. Kaynak, "ALM Tool Infrastructure with a Focus on DevOps Culture", in *Systems, Software and Services Process Improvement*, X. Larrucea, I. Santamaria, R. V. O'Connor, and R. Messnarz, Eds., vol. 896, Cham: Springer International Publishing, 2018, pp. 291–303. DOI: 10.1007/978-3-319-97925-0_24. [Online]. Available: http://link.springer.com/10.1007/978-3-319-97925-0_24 (visited on 01/30/2021).

[S187] J. Baptista, "Agile Documentation with uScrum", in *Proceedings of the 26th Annual ACM International Conference on Design of Communication*, ACM, 2008, pp. 275–276. DOI: 10.1145/1456536.1456596.

[S188] W. Behutiye, P. Karhapää, D. Costal, M. Oivo, and X. Franch, "Non-functional Requirements Documentation in Agile Software Development: Challenges and Solution Proposal", in *International Conference on Product-Focused Software Process Improvement*, M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, and D. Winkler, Eds., vol. 10611, Cham: Springer International Publishing, 2017, pp. 515–522. DOI: 10.1007/978-3-319-69926-4_41. [Online]. Available: http://link.springer.com/10.1007/978-3-319-69926-4_41 (visited on 01/30/2021).

[S190] M. Eckhart and J. Feiner, "How Scrum Tools May Change Your Agile Software Development Approach", in *International Conference on Software Quality*, vol. 238, Cham: Springer International Publishing, 2016, pp. 17–36. DOI: 10.1007/978-3-319-27033-3_2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-27033-3_2.

[S191] G. Goth, "Agile Tool Market Growing with the Philosophy", *IEEE Software*, vol. 26, no. 2, pp. 88–91, 2009. DOI: 10.1109/MS.2009.30.

[S192] E. Hossain, P. L. Bannerman, and D. R. Jeffery, "Scrum Practices in Global Software Development: A Research Framework", in *Proceedings of the 12th International Conference on Product-focused Software Process Improvement*, Springer-Verlag, 2011, pp. 88–102. DOI: 10.1007/978-3-642-21843-9_9.

[S193] M. Isham, "Agile Architecture IS Possible You First Have to Believe!", in *Agile 2008 Conference*, IEEE, Toronto, ON, Canada: IEEE, 2008, pp. 484–489, ISBN: 978-0-7695-3321-6. DOI: 10.1109/Agile.2008.16. [Online]. Available: http://ieeexplore.ieee.org/document/4599526/ (visited on 01/30/2021).

[S194] A. V. K. Prasad, S. Ramakrishna, B. P. Rani, M. U. Kumar, and D. Shravani, "Designing Dependable Business Intelligence Solutions Using Agile Web Services Mining Architectures", in *Information Technology and Mobile Communication*, A. Mobasheri, Ed., Springer, 2011, pp. 301–304, ISBN: 978-3-642-20573-6. DOI: 10.1007/978-3-642-20573-6_51.

[S195] F. Raith, I. Richter, and R. Lindermeier, "How Project-Management-Tools Are Used in Agile Practice: Benefits, Drawbacks and Potentials", in *Proceedings of the 21st International Database Engineering & Applications Symposium*, ACM, 2017, pp. 30–39. DOI: 10.1145/3105831.3105865.

[S196] D. Rost, B. Weitzel, M. Naab, T. Lenhart, and H. Schmitt, "Distilling Best Practices for Agile Development from Architecture Methodology", in *Software Architecture*, D. Weyns, R. Mirandola, and I. Crnkovic, Eds., Springer, vol. 9278, Cham: Springer International Publishing, 2015, pp. 259–267. DOI: 10.1007/978-3-319-23727-5_21. [Online]. Available: http://link.springer.com/10.1007/978-3-319-23727-5_21 (visited on 01/30/2021).

[S197] V. Schneider and R. German, "Integration of Test-Driven Agile Simulation Approach in Service-Oriented Tool Environment", in *Proceedings of the 46th Annual Simulation Symposium*, Society for Computer Simulation International, 2013, p. 7, ISBN: 978-1-62748-030-7. DOI: 10.5555/2499604.2499615.

[S198] S. W. Shin and H. K. Kim, "A Framework for SOA-Based Application on Agile of Small and Medium Enterprise", in *Computer and Information Science*, R. Lee and H.-K. Kim, Eds., vol. 131, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 107–120, ISBN: 978-3-540-79186-7. DOI: 10.1007/978-3-540-79187-4_10. [Online]. Available: http://link.springer.com/10.1007/978-3-540-79187-4_10 (visited on 01/30/2021).

[S199] S. V. Shrivastava and U. Rathod, "A Risk Management Framework for Distributed Agile Projects", *Information and software technology*, vol. 85, pp. 1–15, 2017. DOI: 10.1016/j.infsof.2016.12.005.

[S200] W. Wild, B. Weber, and H. Baumeister, "AOSTA: Agile Open Source Tools Academy", in *International Conference on Agile Processes and Extreme Programming in Software Engineering*, Springer, 2008, pp. 248–249. DOI: 10.1007/978-3-540-68255-4_42.

[S202] E. Knauss, G. Liebel, J. Horkoff, *et al.*, "T-Reqs: Tool Support for Managing Requirements in Large-Scale Agile System Development", in *2018 IEEE 26th International Requirements Engineering Conference (RE)*, IEEE, Banff, AB: IEEE, Aug. 2018, pp. 502–503, ISBN: 978-1-5386-7418-5. DOI: 10.1109/RE.2018.00073. [Online]. Available: https://ieeexplore.ieee.org/document/8491180/ (visited on 01/30/2021).

[S209] R. Nord and J. Tomayko, "Software Architecture-Centric Methods and Agile Development", *IEEE Softw.*, vol. 23, no. 2, pp. 47–53, Mar. 2006, ISSN: 0740-7459. DOI: 10.1109/MS.2006.54. [Online]. Available: http://ieeexplore.ieee.org/document/1605178/ (visited on 01/30/2021).

Summaries

Nederlandstalige Samenvatting

Met de introductie van het Agile Manifesto, Lean Software Development en DevOps wordt er steeds minder gedocumenteerd en is de documentatie van een lagere kwaliteit. Het Agile Manifesto hecht meer waarde aan werkende software dan aan begrijpelijke documentatie, Lean Software Development veronderstelt dat alles wat niet bijdraagt aan klantwaarde opgevat moet worden als verspilling en met DevOps is documentatie beperkt tot gecodeerde infrastructuur om voortdurende wetswijzigingen te volgen en snel de markt op te kunnen.

Naast deze ontwikkelingen valt te onderkennen dat softwareontwikkelaars geen zin hebben om een software product te documenteren en dat softwareontwikkelaars en andere belanghebbenden geen zin hebben om te lezen, kortom TL;DR (te lange tekst, ga ik niet lezen). Dit leidt er toe dat ontwikkelaars telkens tijd moeten investeren om te begrijpen wat de broncode doet en hoe het werkt. Wat niet uit de broncode gehaald kan worden, zijn ontwerpbeslissingen en de rationale waarom de softwareapplicatie werkt als zodanig. Kennisborging betreft dan ook in de eerste plaats ontwerpbeslissingen en rationale.

Om kennisverlies te voorkomen zijn twee hoofdonderzoeksvragen gesteld, waarvan de beantwoording heeft geresulteerd in drie benaderingen met bijbehorende artefacten. De eerste vraag gaat over welke kennis iemand *vooraf* nodig heeft om te beginnen met een project of iteratie. De tweede vraag gaat over welke kennis *achteraf* nodig is om het werk van iemand anders te implementeren, te onderhouden of te gebruiken. De benaderingen heten: ‘Just enough Upfront’, ‘Executable Documentation’ en ‘Automatic Text Analysis’. Met ‘Just enough Upfront’ is het voldoende om schetsen te hebben waarmee een idee gecommuniceerd wordt tussen ontwikkelaars en belanghebbenden. Verder is een stricte gecodificeerde definitie nodig van hoe (sub)systemen met elkaar communiceren en tenslotte is er een plan van aanpak nodig. Karakteristieken van ‘Executable Documentation’ zijn dat ze redelijk leesbare specificaties bevatten van een systeem, en die specificaties kunnen ook uitgevoerd worden. Verder is deze benadering van toepassing als de oplossing van een probleem al goed gespecificeerd is. De laatste benadering betreft ‘Automatic Text Analysis’, waarbij met behulp van machine learning en neurale netwerken causale relaties in tekst en dan met name git commit messages geïdentificeerd worden om ontwerpbeslissingen op te halen. Deze laatste benadering ondersteunt kennisverwerking.

English Summary

With the introduction of the Agile Manifesto, Lean Software Development, and DevOps, documentation has become lower in quality and lesser in quantity leading to knowledge evaporation. The Agile Manifesto values working software over comprehensive documentation, Lean Software Development considers everything that does not contribute to customer value as waste, and in DevOps, documentation is primarily found in infrastructure-as-code to keep up with continuously changing legislation and demands for fast time-to-market.

Next to the aforementioned developments, another issue causing knowledge evaporation concerns the reluctance of software developers to write down information about a software product, and on top of that in general software developers and other stakeholders do not like to read documentation; in short TL;DR. This leads to substantial time investments by new team members to understand what source code intends to do and, foremost, why source code functions the way it does. How source code functions can be distilled from the source code itself. Design decisions or a rationale cannot be revealed from the source code alone. Therefore, knowledge preservation concerns design decisions and rationale in the first place.

To prevent knowledge evaporation, we phrased two main research questions and answers to these questions have led to three approaches with corresponding artifacts. The first question concerns which knowledge someone needs upfront *before* starting a project or iteration. The second question deals with which knowledge *others* need *afterwards* to deploy, maintain, or use a software product. The approaches are 'Just enough Upfront', 'Executable Documentation', and 'Automatic Text Analysis'. 'Just enough Upfront' advocates informal whiteboard sketches to communicate the main objectives between stakeholders, a strict codified interface description between (sub)systems, and a plan of approach. Design decisions are documented afterwards to include progressive insights and deviations from the planned objectives. Characteristics of 'Executable Documentation' concerns human readable requirements and specifications which can be executed. This approach is typically in use when specifications are clearly defined. With 'Automatic Text Analysis', machine learning with neural networks is used to identify causal relations in text, especially git comments, for revealing design decisions. As such, this approach assists in retrieving knowledge.

Curriculum Vitae

Personal	
Name	Theo Theunissen
Date of Birth	2 November 1964
City	Overasselt, NL
Marital status	Married to Judith
Children	Sophie (1997), Sarah (1999)

Professional	
2012 – present	**Stichting HAN University of Applied Sciences** *Function*: Lecturer and researcher in Computer Science *Tasks*: As a computer science teacher for senior students, my responsibilities include coordinating tasks, developing guidelines for multiple disciplines, serving as a member of the quality board, and serving as a member of the advisory board. Additionally, I am an active member of the 'lectoraat', which serves as a bridge between education, research, and professional practice.
2003 - 2012	**Self employed** *Function*: Owner *Tasks*: I held a role as a general manager, responsible for initiating projects, managing customer relationships, and defining start-ups.
1999 - 2003	**IBM Global Services N.V.** *Function*: Senior Consultant Media & Entertainment *Tasks*: Managing / Senior Consultants lead engagements, manage intellectual capital throughout the engagement, participate in business development activities and usually become subject matter experts in an industry, solution and/or methodology. They develop strong, long lasting client relationships and are role models, mentors, and coaches for other team members. All senior Consultant professionals (Executive Consultant, Principal, Managing Principal) must: be thought leaders, be solid practitioners, develop the skills and methodology of their practice, have excellent engagement management skills, establish positive, lasting client relationships
1996 - 1999	**Nielsen Holdings plc (formerly VNU B.V.)** *Function*: Project Manager at corporate office for large and high profile projects, Team leader of project managers, Development Manager, Member of Management Team *Tasks*: These functions demanded my ability to manage projects with high stakes with new technology and complex environments. I was responsible for feasible business plans and project plans, the selection of third parties and the execution of the project.
1994 – 1996	**Oracle B.V.** *Function*: Senior Consultant Advanced Technology *Tasks*: This function demanded my skills of being able to adapt new technology at an early stage, investigate whether this new technology is commercially useful or not and define conditions for commercial usability. I also coached other consultants and skilled them to bring projects back or up to speed.

Education	
1988 - 1994	Philosophy (MA), Cognitive Science, Radboud University Nijmegen

SIKS Dissertations

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring

	32	Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
	33	Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
	34	Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
	35	Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
	36	Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
	37	Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
	38	Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
	39	Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
	40	Christian Detweiler (TUD), Accounting for Values in Design
	41	Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
	42	Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
	43	Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
	44	Thibault Sellam (UVA), Automatic Assistants for Database Exploration
	45	Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
	46	Jorge Gallego Perez (UT), Robots to Make you Happy
	47	Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
	48	Tanja Buttler (TUD), Collecting Lessons Learned
	49	Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
	50	Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains
2017	01	Jan-Jaap Oerlemans (UL), Investigating Cybercrime
	02	Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
	03	Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
	04	Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
	05	Mahdieh Shadi (UVA), Collaboration Behavior
	06	Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
	07	Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
	08	Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
	09	Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
	10	Robby van Delden (UT), (Steering) Interactive Play Behavior
	11	Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
	12	Sander Leemans (TUE), Robust Process Mining with Guarantees
	13	Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
	14	Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
	15	Peter Berck (RUN), Memory-Based Text Correction
	16	Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
	17	Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
	18	Ridho Reinanda (UVA), Entity Associations for Search
	19	Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
	20	Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
	21	Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
	22	Sara Magliacane (VU), Logics for causal inference under uncertainty
	23	David Graus (UVA), Entities of Interest — Discovery in Digital Traces
	24	Chang Wang (TUD), Use of Affordances for Efficient Robot Learning

	25	Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
	26	Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
	27	Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
	28	John Klein (VU), Architecture Practices for Complex Contexts
	29	Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
	30	Wilma Latuny (UvT), The Power of Facial Expressions
	31	Ben Ruijl (UL), Advances in computational methods for QFT calculations
	32	Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
	33	Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
	34	Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
	35	Martine de Vos (VU), Interpreting natural science spreadsheets
	36	Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
	37	Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
	38	Alex Kayal (TUD), Normative Social Applications
	39	Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
	40	Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
	41	Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
	42	Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
	43	Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
	44	Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
	45	Bas Testerink (UU), Decentralized Runtime Norm Enforcement
	46	Jan Schneider (OU), Sensor-based Learning Support
	47	Jie Yang (TUD), Crowd Knowledge Creation Acceleration
	48	Angel Suarez (OU), Collaborative inquiry-based learning
2018	01	Han van der Aa (VUA), Comparing and Aligning Process Representations
	02	Felix Mannhardt (TUE), Multi-perspective Process Mining
	03	Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
	04	Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
	05	Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process
	06	Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
	07	Jieting Luo (UU), A formal account of opportunism in multi-agent systems
	08	Rick Smetsers (RUN), Advances in Model Learning for Software Systems
	09	Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
	10	Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
	11	Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
	12	Xixi Lu (TUE), Using behavioral context in process mining
	13	Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
	14	Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
	15	Naser Davarzani (UM), Biomarker discovery in heart failure
	16	Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
	17	Jianpeng Zhang (TUE), On Graph Sample Clustering
	18	Henriette Nakad (UL), De Notaris en Private Rechtspraak
	19	Minh Duc Pham (VUA), Emergent relational schemas for RDF
	20	Manxia Liu (RUN), Time and Bayesian Networks

21 Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araújo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (UVT), Emotion Recognition from Mandarin Speech
30 Wouter Beek, The "K" in "semantic web" stands for "knowledge": scaling semantics to the web

---

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TUE), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TUE), Process Mining with Streaming Data
06 Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UVA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VU), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TUE), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TUE), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VU),Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VU), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VU), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OUN), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
29 Daniel Formolo (VU), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VU), Alive and Kicking: Baby Steps in Robotics
32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games

33 Anil Yaman (TUE), Evolution of Biologically Inspired Learning in Artificial Neural Networks
34 Negar Ahmadi (TUE), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OUN), Gamification with digital badges in learning programming
36 Kevin Ackermans (OUN), Designing Video-Enhanced Rubrics to Master Complex Skills
37 Jian Fang (TUD), Database Acceleration on FPGAs
38 Akos Kadar (OUN), Learning visually grounded and multilingual representations

---

2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
04 Maarten van Gompel (RUN), Context as Linguistic Bridges
05 Yulong Pei (TUE), On local and global structure mining
06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
07 Wim van der Vegt (OUN), Towards a software architecture for reusable game components
08 Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search
09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
10 Alifah Syamsiyah (TUE), In-database Preprocessing for Process Mining
11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data AugmentationMethods for Long-Tail Entity Recognition Models
12 Ward van Breda (VU), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
15 Konstantinos Georgiadis (OUN), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
17 Daniele Di Mitri (OUN), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
20 Albert Hankel (VU), Embedding Green ICT Maturity in Organisations
21 Karine da Silva Miras de Araujo (VU), Where is the robot?: Life as it could be
22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
24 Lenin da Nóbrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
25 Xin Du (TUE), The Uncertainty in Exceptional Model Mining
26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
30 Bob Zadok Blok (UL), Creatief, Creatieve, Creatiefst
31 Gongjin Lan (VU), Learning better – From Baby to Better
32 Jason Rhuggenaath (TUE), Revenue management in online markets: pricing and online advertising
33 Rick Gilsing (TUE), Supporting service-dominant business model evaluation in the context of business model innovation
34 Anna Bon (MU), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production

---

2021 01 Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space

02 Rijk Mercuur (TUD), Simulating Human Routines: Integrating Social Practice Theory in Agent-Based Models
03 Seyyed Hadi Hashemi (UVA), Modeling Users Interacting with Smart Devices
04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
07 Armel Lefebvre (UU), Research data management for open science
08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children's Collaboration Through Play
10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
16 Esam A. H. Ghaleb (UM), Bimodal emotion recognition from audio-visual cues
17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks
18 Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
19 Roberto Verdecchia (VU), Architectural Technical Debt: Identification and Management
20 Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
21 Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
22 Sihang Qiu (TUD), Conversational Crowdsourcing
23 Hugo Manuel Proença (LIACS), Robust rules for prediction and description
24 Kaijie Zhu (TUE), On Efficient Temporal Subgraph Query Processing
25 Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
26 Benno Kruit (CWI & VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
27 Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
28 Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs

---

2022 01 Judith van Stegeren (UT), Flavor text generation for role-playing video games
02 Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey
03 Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
04 Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
05 Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
06 Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding
07 Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
08 Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
09 Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
10 Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
11 Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
12 Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
13 Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
14 Michiel Overeem (UU), Evolution of Low-Code Platforms