Complex Systems Informatics and Modeling Quarterly (CSIMQ)
eISSN: 2255-9922
Published online by RTU Press, https://csimq-journals.rtu.lv
Article 178, Issue 32, September/October 2022, Pages 1–27
https://doi.org/10.7250/csimq.2022-32.01

Approaches for Documentation in Continuous Software Development

Theo Theunissen[1⋆], Stijn Hoppenbrouwers[1,2], and Sietse Overbeek[3]

[1]Department of ICT, HAN University of Applied Sciences, Arnhem, the Netherlands
[2]Radboud University, Institute for Computing and Information Sciences, Nijmegen, the Netherlands
[3]Department of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands
theo.theunissen@han.nl, stijn.hoppenbrouwers@han.nl, s.j.overbeek@uu.nl

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.

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

⋆ Corresponding author

Reference: T. Theunissen, S. Hoppenbrouwers, and S. Overbeek, "Approaches for Documentation in Continuous Software Development," Complex Systems Informatics and Modeling Quarterly, CSIMQ, no. 32, pp. 1-–27, 2022. Available: https://doi.org/10.7250/csimq.2022-32.01

Additional information. Author ORCID iD: T. Theunissen – https://orcid.org/0000-0003-0681-8666, S. Hoppenbrouwers – https://orcid.org/0000-0002-1137-2999, and S. Overbeek – https://orcid.org/0000-0003-3975-200X. PIIS225599222200178X. Received: 30 May 2022. Revised: 1 September 2022. Accepted: 1 September 2022. Online available: 28 October 2022.

becomes more critical. However, the attention span for documentation reading, in general, has decreased dramatically [1]. In previous research [2], [3], we observed that developers do not want to write, others do not want to read, but having no 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 article, we will address three possible approaches that contribute to answering this question in the context of Continuous Software Development (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 (Figure 1). 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 2, we will describe the conceptual research framework

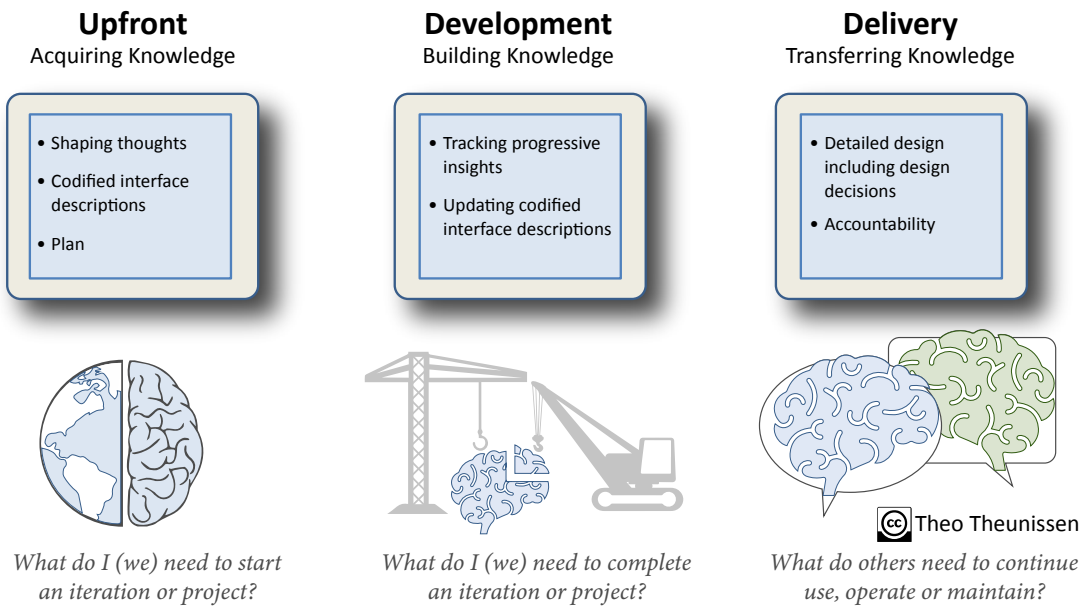

Figure 1. Phases with Knowledge Acquisition, Knowledge Building, and Knowledge Transfer

for constructing[4] 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 1 for a diagram. When we refer to 'knowledge', we refer to all *types of information* as shown in Figure 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

[4] 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.

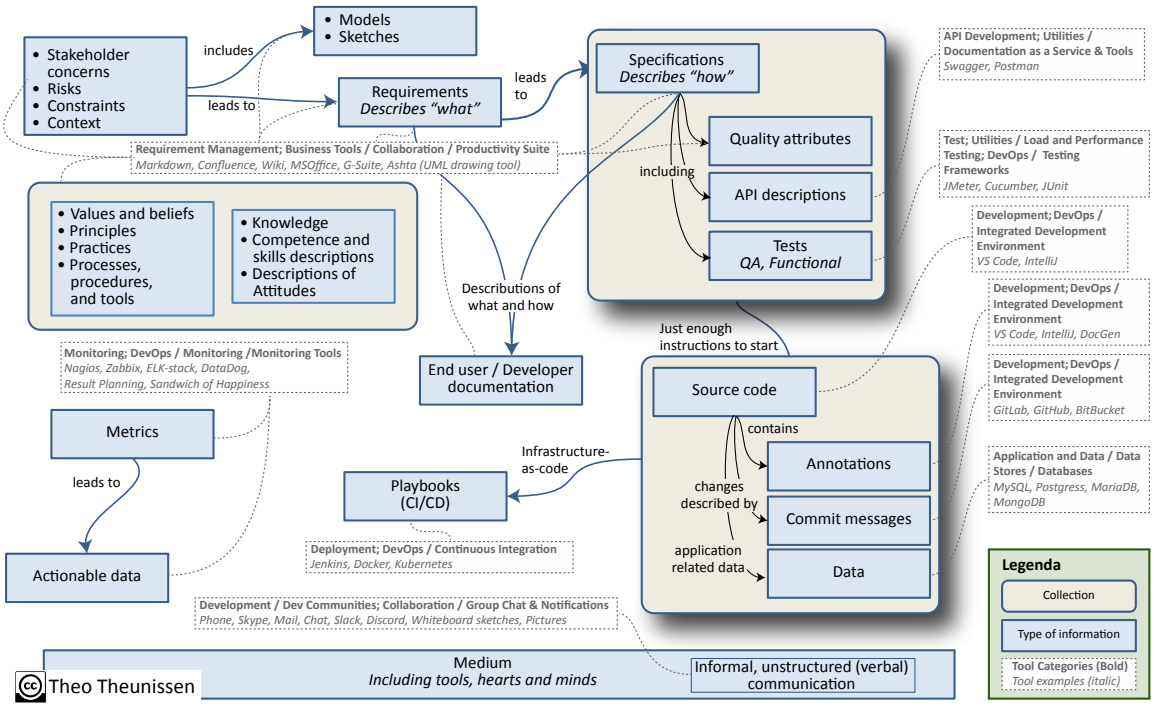

Figure 2. Types of Information including mapping to tool categories and tools [3]

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.

1.1 Previous Research

This study follows up on previous research, where we have found three candidate approaches. Figure 3 shows the studies in this research project and results from previous studies. The approaches in this study are the elaborated recommendations from a previous systematic mapping study [2]. 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 [3]. 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.

1.2 Contributions

The contributions of this article 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., [4], but have not made a clear and sharp distinction between artifacts that are of typical in use upfront, during, or afterward. Furthermore, using Natural Language Processing (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, Rational Unified Process (RUP), because of efficiency and pragmatic reasons that work well for them. However, this way of working is not always validated or supported by management, and is often not included in curricula. 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 article, the following subjects are addressed. In Section 2, the research questions and research design are described. The approaches are introduced in Section 3. Section 3.1 describes the 'Just enough Documentation' approach. Next, in Section 3.2, 'Executable Documentation' is explained, and in Section 3.3 the approach with 'Automated Text Analysis' is described. In Section 4, the Threats to Validity are discussed. Finally, conclusions and future research are described in Section 5.

2 Research Design

2.1 Research Questions

The research questions are defined as documentation-related questions, which incorporate knowledge questions. The proposed approaches follow a previous systematic mapping study [2] and a case study [3], and fit within the research cycle of a literature review [3], field research [2], the construction of approaches (this article), and finally a validation of the approaches (future research). In Figure 3, the phases are depicted.

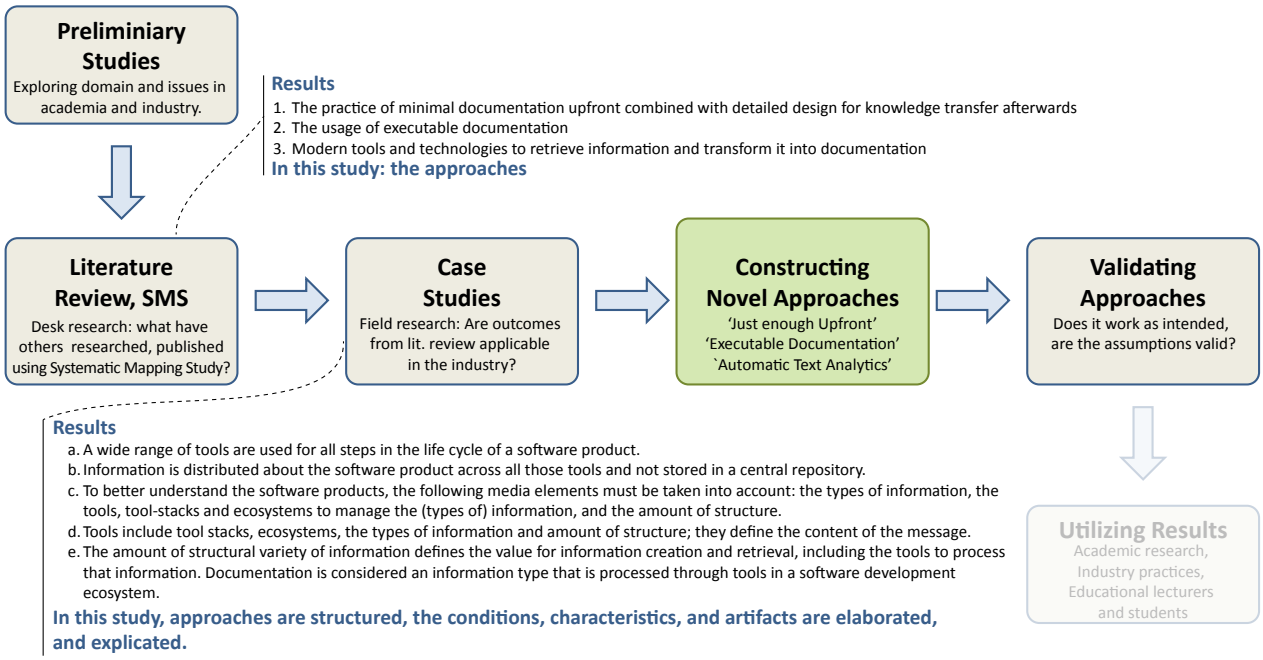

Figure 3. Studies in this research project

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?

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 4.

The dynamic part depicts a flow that starts from discovering phenomena that define the problem and ends with the invention of a solution. The discovery of phenomena refers to observations in a platonic (εἶδος, 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.[5] 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 [5]. Researchers try to establish logical and

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

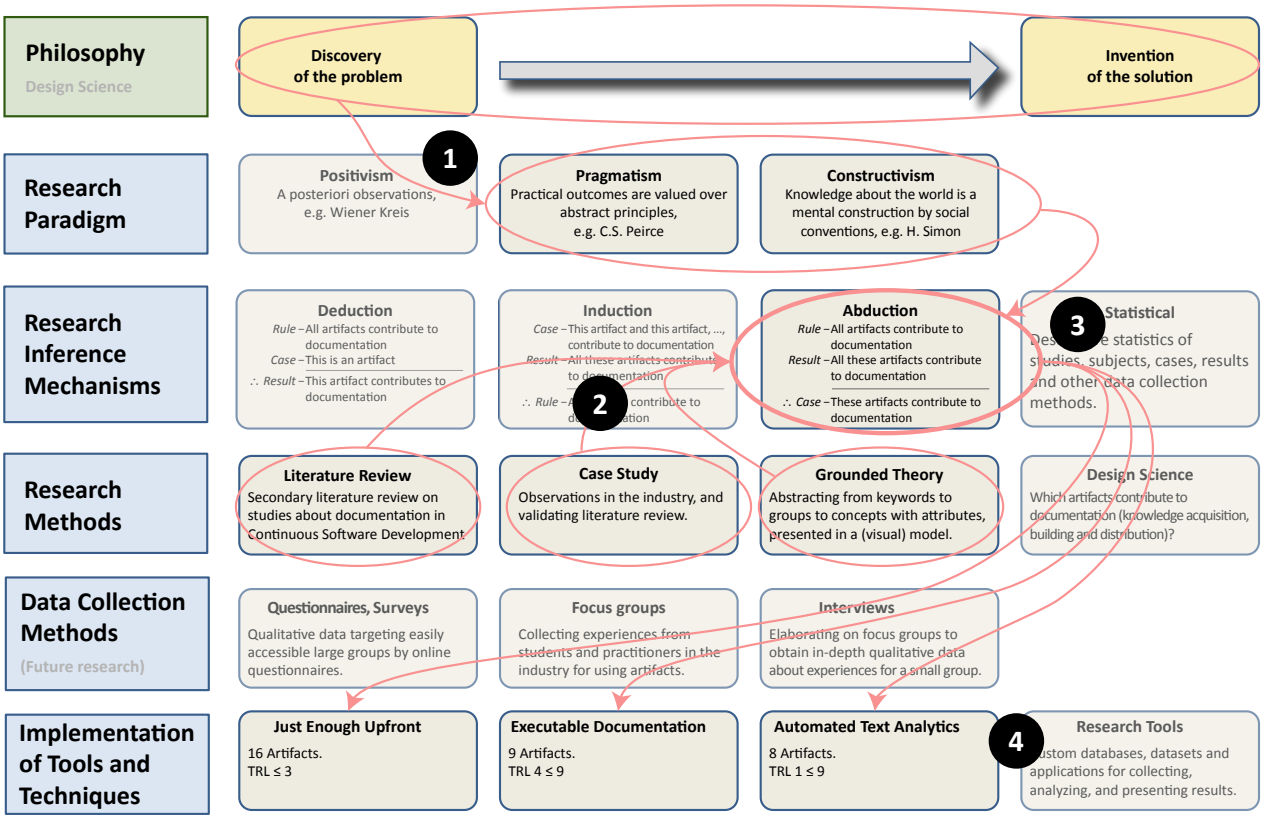

Figure 4. The Conceptual Research Framework

causal relations between phenomena with inference mechanisms. With 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 [6]. 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 Kitchenham and Charters [7], [8] 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) [9]. With grounded theory Stevens *et al.* [10] 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 (SD) based on March and Smith [11], [12], [13], [14], 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.

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 2.2. Design Science is used for discovering the problem and for invention of the solution [11]. A typical research, according to Wieringa [14], cycle includes a

problem investigation, treatment design, treatment validation and treatment implementation [14]. From the research paradigms, constructivism is used [12]. A typical inference method for generating a hypothesis, i.e., approaches, is abduction [6]. A systematic mapping study and validation in the industry with case studies leading to a grounded theory have been conducted in previous studies [2], [3]. 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, were used in previous studies [3] 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 [15], [2], [3], 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 [16].

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[6]. 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 [3]. 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 1.

The approaches are explored by applying the conceptual research framework (see Section 2.2 and Figure 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 Level (TRL) [17]. In Figure 4, (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; and in (4) the approaches are presented.

3.1 'Just Enough Upfront' Documentation to Start a Software Product, a Project or an Iteration

This approach results from a prior study [2]. The conditions, characteristics, and artifacts are elaborated, explicated, and structured for this approach. When using this approach, the bare

[6] https://cucumber.io/

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 1, the artifacts are presented that are in use with this approach, as derived from previous research [2]. They will be discussed below.

Table 1. Phases, Processes, and Artifacts in Continuous Software Development

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.

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 Technology Readiness Levels (TRL) lower than or equal to three where an idea must be validated in a Proof of Concept (PoC) [17]. It fits within the Agile philosophy that working software is valued over comprehensive documentation [18]. It follows the Lean principle that anything that does not contribute to the end-product is considered waste [19].

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. Keys 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 2 for an overview of types of documents. The audience for this overview comprises all stakeholders. There is no specific template. Typical tools are web pages as a starting point from, e.g., 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 [20] 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 Unified Modelling Language (UML) is not actually required [2], [3]. Basically, any box-and-line diagram that conveys an idea and achieves mutual understanding will do [21]. Ainsworth [22] 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 [23], [24]. The following aspect characteristics are based on Agile decision making as discussed by Rouse [25].

1. *Priorities.* Refer to the primary objectives for the software product. This includes a description of what criteria determine the 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[7], benchmark sites such as databases[8], jobs[9], trending technologies on Gartner hype cycle[10], and Thoughtworks[11]. Other aspects such as economic, legal, social, environmental, etc., are valid as well [26]. The number of items on the long list varies between 15 to 25, depending on the context.
3. *Shortlist.* Make clear what defines the scope of relevant techniques. The number of items on the shortlist varies between 3 and 5, depending on the context. A shortlist with only one item and no alternatives is not convincing.
4. *Features of a 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), Simple Object Access Protocol (SOAP), JavaScript Object Notification (JSON) or XML-RPC, and technologies such as Common Object Request Broker Architecture

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

(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) websites.

D6. *Plan of Approach*. Elaborating on the systematic review by Abrahamsson *et al.* [27] and empirical research by Dybå and Dingsøyr [28], 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: a lack of insight and a lack of control. The lack of insights refers to situations where decisions are made without sufficient information. The lack of control refers to situations where can not be carried 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[12] or Jira[13] 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 items 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 communicating concepts such as values, beliefs, mission, vision, strategy, and objectives will do. Commonly in use for developers in CSD are 4+1 from Kruchten [29] and C4 from Brown [30]. Both 4+1 and C4 are architectural views with multiple models. For 4+1, these models are the logical view, development view, physical view, process view, and the (use case) scenarios. For C4, the models concern the context, containers, components, and classes. The audience are 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 [7] and a systematic mapping study Petersen *et al.* [8] 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 *et al.* [31] 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 [32]. 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.

[12] https://trello.com/

[13] https://www.atlassian.com/software/jira

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 [32]. This can range from a commitment to results, personal thoughts, social aspects, or any other factors that had an effect. The primary audience are coaches and developers. An extended description and template can be found at https://theotheunissen.nl/happiness.

D11. *Software*. This refers to executable files including source code, scripts, and executable specifications such as Continuous Integration/Continuous Deployment (CI/CD), Data Definition Language (DDL), or Data Manipulation Language (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 are the development team, including new team members. Templates can be found at 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. 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 are stakeholders, especially the development team, including new team members. Any template can be used that follows the traditional Software Requirements Specifications [33], SAD [34] and Software Design Descriptions [35]. Also, C4 [30] or 4+1 from Kruchten [29] can serve as templates.

D14. *Decisions*. With decisions, an assessment is made based on argumentation. 'Forces' influence the decisions and can be a trade-off. For instance, user-friendliness and safety are sometimes contrary forces. Stakeholders make decisions, and the audience are the stakeholders in a category for which the force is relevant, e.g., customers, developers, end-users, and 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[14], Architecture Decision Record (ADR)s[15], or Decision Centric Architecture Review (DCAR) [36].

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) [37]. 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.

[14] https://wiki.c2.com/?PatternCommunity

[15] https://adr.github.io/

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 replaced by a throw-away document and a document at the end of a project or iteration that transfers knowledge about the software product.

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 2, artifacts related to ED are listed. Furthermore, ED can be tested because it can be executed.

Table 2. Phases, Processes, and Artifacts in Continuous Software Development. All artifacts refer to executable code.

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, Application Programming Interface (API)s.
E4		Quality control.	•Tests (Test Driven Development (TDD), Behavior Driven Development (BDD), Acceptance Test Driven Development (ATDD)).
E5		Management.	• Definition of Done, Acceptance Criteria, Specific, Measurable, Acceptable, Relevant, Time-bound (SMART) Key Performance Indicator (KPI)s.
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.

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 1, knowledge acquiring, knowledge building, and distributing knowledge also applies to executable documentation.

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.

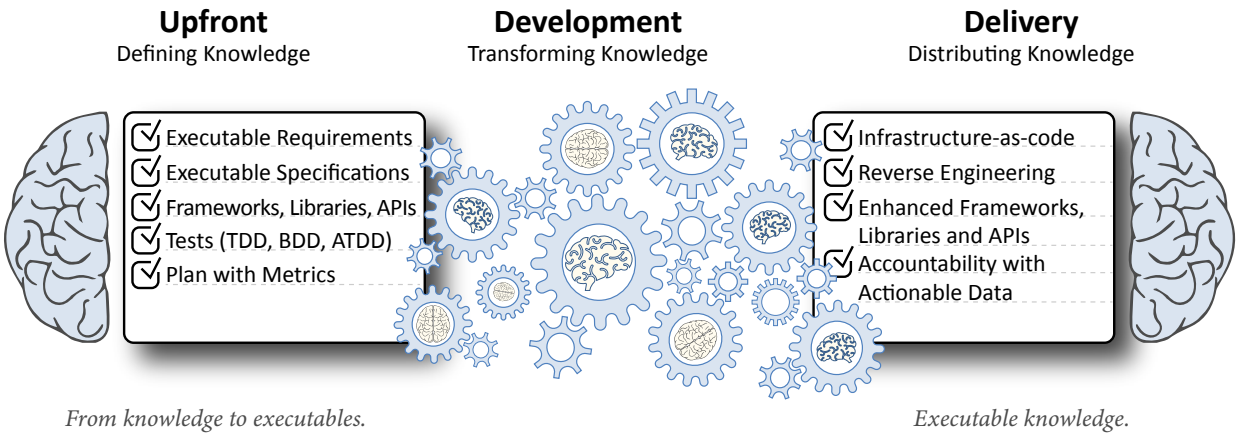

Figure 5. Executable documentation in consecutive phases

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 Time-to-Market (TTM) is typical for projects that need to be ahead of the competition and keep pace with legislation [38].

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 [39]. 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[16], React Native[17]) or Object Relational Mapping (ORM)

[16] http://angular.io/

[17] https://reactnative.dev/

frameworks such as Hibernate[18] or Sequelize[19]). 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 [40]. 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).* The tests are represented in Figure 6. An arbitrary categorization for tests is following the categories as mentioned in ISO/IEC-25010 [41]. This includes functional and non-functional categories such as scalability and security.

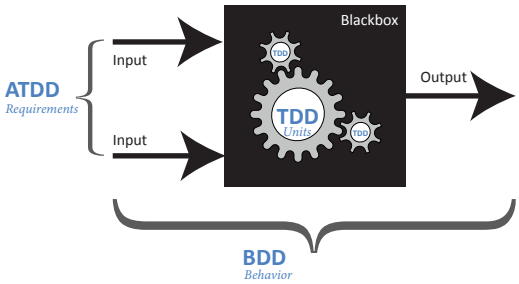

Figure 6. TDD, BDD, and ATDD and where they do apply to a software product

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 [42], [43], [44].

BDD is is the testing of the behavior of the system, based on related input and expected output[20] [45], [46].

ATDD refers to capturing and validating requirements by analyzing user stories [47], [48], [49].

E6. *Infrastructure-as-code.* This concept is about the automated testing, integration, deployment, and delivery of source code, commonly in use in the CI/CD and DevOps communities [50]. 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 Yet Another Markup Language (YAML)[21], a relatively easy-to-read mark-up language. Examples for provisioning a system are Ansible[22] or GitOps [51].

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. 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.

[18] https://hibernate.org/
[19] https://sequelize.org/
[20] https://www.behaviourdriven.org/
[21] https://yaml.org/
[22] https://www.ansible.com

E8. *Enhanced Templates, Frameworks, Libraries, APIs.* The enhancements refer to the new iteration of the template, framework, library, or API. Nah *et al.* [52] and Ghezzi [53] 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 *et al.* [32] concerning the Result Planning and Sandwich of Happiness, it does not make sense to have a plan of approach without taking accountability. Also, it does not make sense to take accountability when there was no planning to compare the outcomes of outlined actions.
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 [54]. 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 specifications 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, and 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 replaced by upfront throw-away documentation. The knowledge required by system engineers, end-users, and maintainers is created afterwards. In terms of TRL, this applies to a prototype and pilot, which are levels higher than four.

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 instance, 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 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.

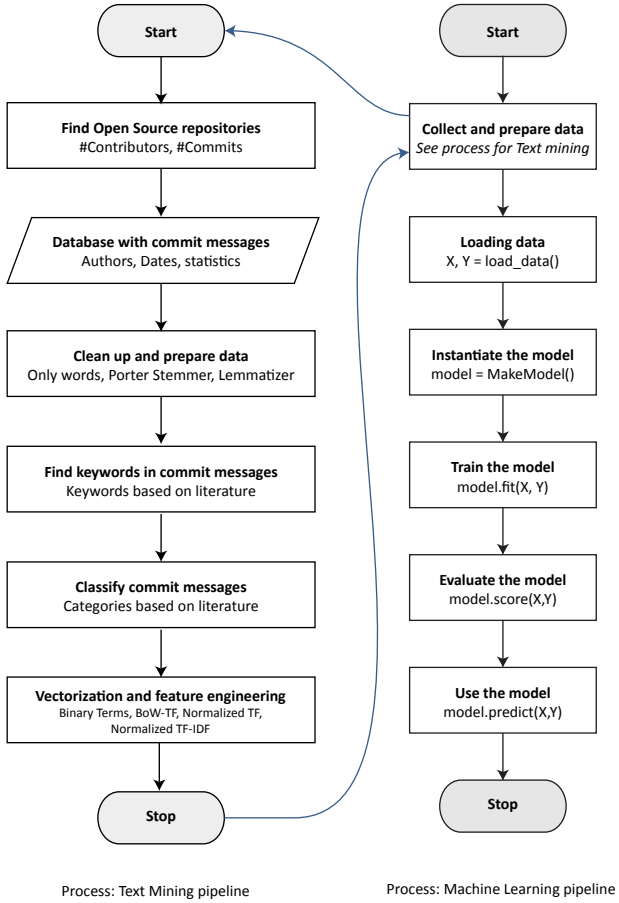

Figure 7. Pipelines for Text Mining and Machine Learning

Table 3. Candidate Artifacts in Automated Text Analytics

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.

After the 'AI Winters', occurring because of failure to deliver on promises [55], [56], ending around 1993, a range of neural networks came into existence such as Convolutional Neural Network (CNN)s, [57] for image processing and Recurrent Neural Network (RNN)s [58] for language processing, visually beautifully explained by Veen [59]. Figure 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.

In Table 3, the artifacts are shown that are in use with this approach.

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 [60]. 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) [61].

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 *et al.* [62] 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 'Automated Text Analytics' 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 *et al.* [63] 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 4, commit types based on *Conventional Commits* [60] are shown.

Table 4. Conventional commit types for Git. The most relevant are Features and Bug Fixes

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

A3. *Statistics: BoW, TF, IDF, TFIDF.* Table 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 [64]. 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.

Table 5. Statistical methods for NLP and Text Mining

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.	$tf(t,d) = \frac{f_{t,d}}{\sum\limits_{t^{'} \in d} f_{t^{'},d}}$
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.	$tdidf(t,d,D) = tf(t,d) \times idf(t,D)$

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.

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 [65]. 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 [66] already in 1976 introduced transfer learning. An example for NLP is BERT [67] 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 6.

A8. *Finding Design Decisions in different Types of Information.* The objective of the types of information (Figure 2) is to convey an understanding for *why* decisions are taken for the specific

Table 6. Difference between model parameters and hyperparameters

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.

software design. 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 the approach discussed in this Section. The approach counts the eight most prominent artifacts with annotated data, model, hyperparameters, and transfer learning among them. The tool for source management control, i.e., Git, is relevant because it is 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.

4 Threats to Validity

For the previous studies ([2], [3]), the threats were addressed as follows. The initial search process identified threats concerning study selection, where the set of candidate papers for primary studies was selected, and the study filtering, where the final set of primary studies was 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 [14] defines the following threats to validity:

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 a systematic mapping study, interviews, and a case study. The data consist of literature, non-executable artifacts such as Git, the Atlassian[23] stack and other documents.
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 it is applicable in other domains than were the cases originate. This threat is not applicable because we focus on the domain of software engineering and not on other domains.
4. Construct validity is the degree to which inferences from phenomena to construct are justified. In this study this applies to the abductive reasoning process from findings in the systematic mapping study and observations in case studies. The result of the reasoning process is not evaluated in this study. The positivist result would be a falsification for one of the approaches and artifacts.
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.

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 the research questions are displayed in relation to each other, included in the 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 a software product. Table 7 shows the answers in brief to the research questions. Common characteristics across the approaches ‘Just enough Upfront’, ‘Executable Documentation’, and ‘Automated Text Analytics’ concern 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

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

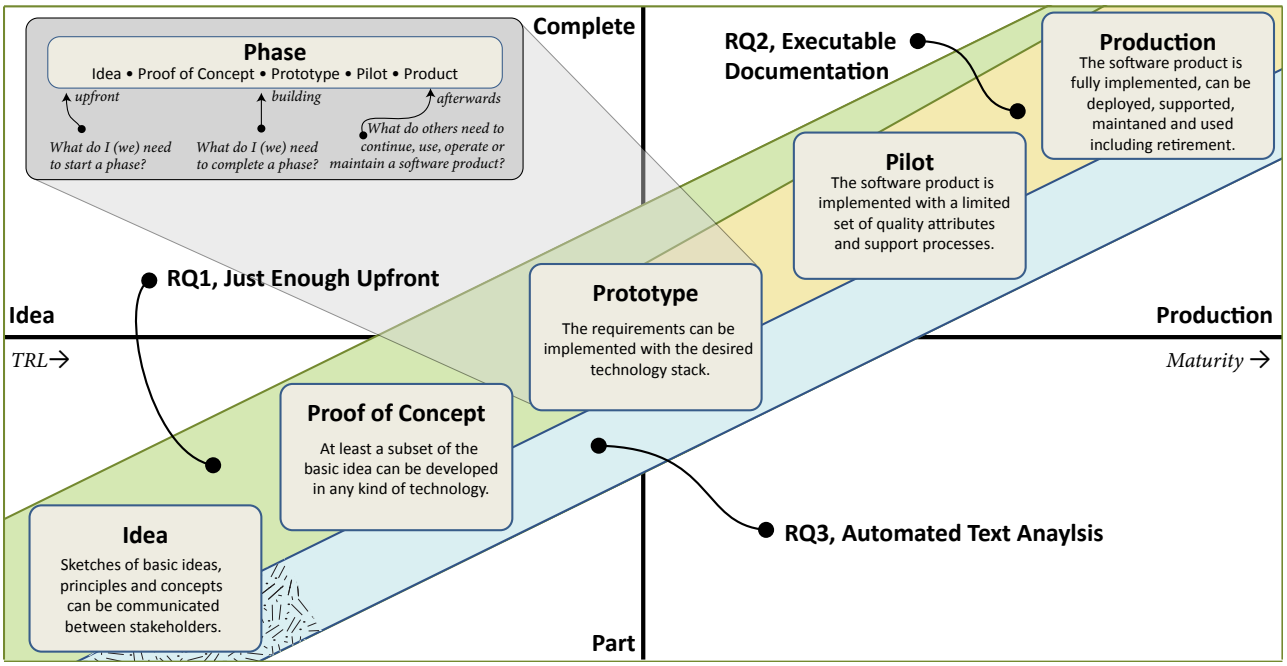

Figure 8. Relation between research questions, maturity, and completeness of the software product for each phase from Idea to Production

sixteen artifacts identified, of which the most relevant are upfront: whiteboard sketches, a codified interface description, and a 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 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.

5.1 Discussion on how the Approaches were Constructed

It is not possible to observe such relations as 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 *et al.* [68]. Second, 'constructivism' is the paradigm that March and Smith [11] consider to be applicable for design science.

Table 7. Answers to Research Questions

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

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 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 help to support or reject correlation.

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 messages 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.

References

[1] K. R. Subramanian, "Myth and Mystery of Shrinking Attention Span", *International Journal of Trend in Research and Development*, vol. 5, no. 3, pp. 1–6, 2018. [Online]. Available: http://www.ijtrd.com/papers/IJTRD16531.pdf.

[2] 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. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106733.

[3] 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. [Online]. Available: https://doi.org/10.5220/0010367901530164.

[4] 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, 2018. [Online]. Available: https://doi.org/10.1186/s40411-018-0051-7.

[5] T. Kuhn, *The Structure of Scientific Revolutions*. Princeton University Press, 1970.

[6] 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/.

[7] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering", *Engineering*, vol. 2, p. 1051, 2007. [Online]. Available: http://cdn.elsevier.com/promis_misc/525444systematicreviewsguide.pdf.

[8] 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, 2008. [Online]. Available: https://doi.org/10.14236/ewic/EASE2008.8.

[9] R. Yin, *Case Study Research*, Fourth Edition. Thousand Oaks, CA: Sage Publications, 2008.

[10] 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.

[11] 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. [Online]. Available: https://doi.org/10.1016/0167-9236(94)00041-2.

[12] H. A. Simon, *The Sciences of the Artificial*. Cambridge, Massachusetts: MIT Press, 1996. [Online]. Available: https://doi.org/10.7551/mitpress/12107.001.0001.

[13] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research", *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004. [Online]. Available: https://dl.acm.org/doi/10.5555/2017212.2017217.

[14] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, 2014. [Online]. Available: https://doi.org/10.1007/978-3-662-43839-8.

[15] T. Theunissen and U. Van Heesch, "Specification in Continuous Software Development", in *Proceedings of the 22ND European Conference on Pattern Languages of Programs*, ser. EuroPLoP '17, ACM, Association for Computing Machinery, 2017, pp. 1–19. [Online]. Available: https://doi.org/10.1145/3147704.3147709.

[16] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st. New York: Addison-Wesley Professional, 2015.

[17] B. Dunbar. "Technology Readiness Level", NASA. (2021), [Online]. Available: https://www.nasa.gov/directorates/heo/scan/engineering/technology/technology_readiness_level.

[18] K. Beck *et al.* "Manifesto for Agile Software Development Twelve Principles of Agile Software", Manifesto for Agile Software Development. (2001), [Online]. Available: https://agilemanifesto.org/.

[19] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*. Boston, MA: Addison-Wesley Educational, 2003.

[20] C. E. Shannon and W. Weaver, "The Mathematical Theory of Communication", *Urbana: University of Illinois Press*, 1949.

[21] G. Sperling, “The Information Available in Brief Visual Presentations.”, *Psychological monographs: General and applied*, vol. 74, no. 11, pp. 1–29, 1960. [Online]. Available: https://doi.org/10.1037/h0093759.

[22] S. Ainsworth, “DeFT: A Conceptual Framework for Considering Learning With Multiple Representations”, *Learning and instruction*, vol. 16, no. 3, pp. 183–198, 2006. [Online]. Available: https://doi.org/10.1016/j.learninstruc.2006.03.001.

[23] 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. [Online]. Available: https://doi.org/10.1016/j.jss.2012.01.058.

[24] 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. [Online]. Available: https://doi.org/10.1016/j.infsof.2011.11.006.

[25] W. B. Rouse, “Agile Information Systems for Agile Decision Making”, in *Agile Information Systems*, K. C. DeSouza, Ed., London: Routledge, 2007, pp. 16–30.

[26] J. V. Richardson Jr. “STEPE – Social, Technical, Economic, Political and Ecological Factor Model”. (1990), [Online]. Available: https://pages.gseis.ucla.edu/faculty/richardson/STEPE.htm.

[27] 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. [Online]. Available: https://doi.org/10.1109/ICSE.2003.1201204.

[28] 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, 2008. [Online]. Available: https://doi.org/10.1016/j.infsof.2008.01.006.

[29] P. Kruchten, “The 4+1 View Model of Architecture”, *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, 1995. [Online]. Available: https://doi.org/10.1109/52.469759.

[30] S. Brown, *Software Architecture for Developers*. 2014, p. 233. [Online]. Available: https://leanpub.com/software-architecture-for-developers.

[31] 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, 2015. [Online]. Available: https://doi.org/10.1016/j.jss.2014.09.042.

[32] 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. [Online]. Available: https://doi.org/10.1145/3489449.3489974.

[33] Standards Committee, “ISO/IEC/IEEE international standard – systems and software engineering – life cycle processes – Requirements engineering”, *ISO/IEC/IEEE 29148:2011(E)*, pp. 1–94, 2011. [Online]. Available: https://doi.org/10.1109/IEEESTD.2011.6146379.

[34] Technical Committee, “ISO – 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.

[35] 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, IEEE, 2009. [Online]. Available: https://doi.org/10.1109/IEEESTD.2009.5167255.

[36] 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. [Online]. Available: https://doi.org/10.1109/MS.2013.22.

[37] E. Ries. "Minimum Viable Product: A guide". (2009), [Online]. Available: http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html.

[38] 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. [Online]. Available: https://doi.org/10.1287/mnsc.42.2.173.

[39] K. E. W. Morand, "Software Requirements As Executable Code", Regis University, Dayton Memorial Library, Master Thesis, 2012. [Online]. Available: https://epublications.regis.edu/theses/232/.

[40] A. Silva *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. [Online]. Available: https://doi.org/10.1145/3084226.3084262.

[41] 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. [Online]. Available: https://www.iso.org/standard/35733.html.

[42] D. North *et al.* "Introducing BDD", Better Software, March. (2006), [Online]. Available: https://dannorth.net/2006/10/20/article-introducing-behaviour-driven-development/.

[43] 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. [Online]. Available: https://doi.org/10.1109/MS.2010.152.

[44] 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. [Online]. Available: https://doi.org/10.1145/3382494.3410687.

[45] 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. [Online]. Available: https://doi.org/10.1109/SEAA.2011.76.

[46] 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. [Online]. Available: https://doi.org/10.1109/ICGSE.2019.00016.

[47] K. Pugh, *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Boston, MA: Pearson Education, 2010.

[48] 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, 2010, pp. 208–213. [Online]. Available: https://doi.org/10.1007/978-3-642-13054-0_20.

[49] 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. [Online]. Available: https://doi.org/10.1145/3335595.3335650.

[50] M. Hüttermann, "Infrastructure as Code", in *DevOps for Developers*, Springer, 2012, pp. 135–156. [Online]. Available: https://doi.org/10.1007/978-1-4302-4570-4_9.

[51] F. Beetz and S. Harrer, "GitOps: The evolution of DevOps?", *IEEE Software*, vol. 39, no. 4, pp. 70–75, 2022. [Online]. Available: https://doi.org/10.1109/MS.2021.3119106.

[52] 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. [Online]. Available: https://doi.org/10.1002/smr.239.

[53] C. Ghezzi, "Of Software and Change", *Journal of Software: Evolution and Process*, vol. 29, no. 9, e1888, 2017. [Online]. Available: https://doi.org/10.1002/smr.1888.

[54] R. Cross and L. Sproull, "More Than an Answer: Information Relationships for Actionable Knowledge", *Organization science*, vol. 15, no. 4, pp. 446–462, 2004. [Online]. Available: https://doi.org/10.1287/orsc.1040.0075.

[55] 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.

[56] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach, Global Edition*, 4th ed. London, England: Pearson Education, 2021.

[57] 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. [Online]. Available: https://doi.org/10.1113/jphysiol.1962.sp006837.

[58] 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. [Online]. Available: https://doi.org/10.7551/mitpress/5236.001.0001.

[59] F. V. Veen. "The Neural Network Zoo", The Asimov Institute. (2016), [Online]. Available: https://www.asimovinstitute.org/neural-network-zoo/.

[60] "Conventional Commits", Conventional Commits. (2022), [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/.

[61] 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.

[62] 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. [Online]. Available: https://doi.org/10.1007/s10115-022-01665-w.

[63] 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. [Online]. Available: https://doi.org/10.1145/3266237.3266260.

[64] M. F. Porter, "An Algorithm for Suffix Stripping", *Program: electronic library and information systems*, vol. 14, no. 3, pp. 130–137, 1980. [Online]. Available: https://doi.org/10.1108/eb046814.

[65] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016. [Online]. Available: http://www.deeplearningbook.org.

[66] S. Bozinovski, "Reminder of the First Paper on Transfer Learning in Neural Networks, 1976", *Informatica*, vol. 44, no. 3, 2020. [Online]. Available: https://doi.org/10.31449/inf.v44i3.2828.

[67] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1810.04805.

[68] 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.