

# Specification in Continuous Software Development

THEO THEUNISSEN and UWE VAN HEESCH, HAN University of Applied Sciences

---

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.

CCS Concepts: •**Software and its engineering** → **Patterns**; *Designing software*;

Additional Key Words and Phrases: Lean, Agile, DevOps, Continuous Development, Software engineering

## ACM Reference Format:

Theo Theunissen and Uwe van Heesch. 2017. Specification in Continuous Software Development. *EuroPLoP* (July 2017), 19 pages.

DOI: <https://doi.org/10.1145/3147704.3147709>

---

## 1. INTRODUCTION

In our previous research, we have been investigating practices in design and documentation in web and mobile applications [Theunissen and van Heesch 2016]. 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 [Poppendieck and Poppendieck 2003] is to avoid efforts that do not increase value for the customer. Agile software development [Fowler and Highsmith 2001] 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 [Erich et al. 2014], development teams are formed in way that

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

EuroPLoP '17, July 12-16, 2017, Irsee, Germany

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-4848-5/17/07...\$15.00

<https://doi.org/10.1145/3147704.3147709>

the 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 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 end-point (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.

## 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.

### 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 2003], 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 losing 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.

## 2.2 Agile

In 2001, the agile manifesto [Fowler and Highsmith 2001] gave rise to a new way of thinking and collaborating in software projects. Since then, several process models (the most prominent being scrum [Alliance 2017] and XP [Beck 1999]) 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.

### 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 [Erich et al. 2014].

- (1) **Culture** The primary characteristic of a DevOps culture is *increased collaboration* between the roles of development and operations [Wilsenach 2016]. 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 2013] emphasizes *open communication, alignment of incentives and responsibilities, respect*, and finally, *trust*.
- (2) **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.
- (3) **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.
- (4) **Sharing** In DevOps, sharing refers to knowledge, tools and successes [Humble and Molesky 2011]. 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.
- (5) **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 [Lovelock and Gummesson 2004].
- (6) **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 [Cukier 2013].
- (7) **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.

### 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 prac-

tices. In the remainder of this paper, we will refer to development processes that exhibit these shared principles as *Continuous Software Development (ConSD)*.

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 [Brooks Jr 1995]<sup>1</sup>. 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 time-to-market, 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.

<sup>1</sup>“If one woman delivers a baby in nine months, then nine women can't deliver a baby in one month”

- (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 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. PATTERN: CONTINUOUS SOFTWARE DESIGN SPECIFICATION

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

#### 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<sup>2</sup> or Gradle<sup>3</sup>), a distributed version control system (like Git<sup>4</sup>), a document management system and wiki (for instance Confluence<sup>5</sup>), and a task and project tracking tool (see for instance Jira<sup>6</sup>).

#### 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?**

<sup>2</sup><https://maven.apache.org>

<sup>3</sup><https://gradle.org>

<sup>4</sup><https://git-scm.com>

<sup>5</sup><https://www.atlassian.com/software/confluence>

<sup>6</sup><https://www.atlassian.com/software/jira>

### 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.
- (5) **Overestimated competences and underestimated complexity** People tend to overestimate their own competences and skills [Kruger and Dunning 1999]. 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.

### 3.4 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:**

**Specifications at the start of an iteration →**

### **Specifications during an iteration → Refactored specifications at the end of an iteration**

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.

3.4.1 *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 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 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:



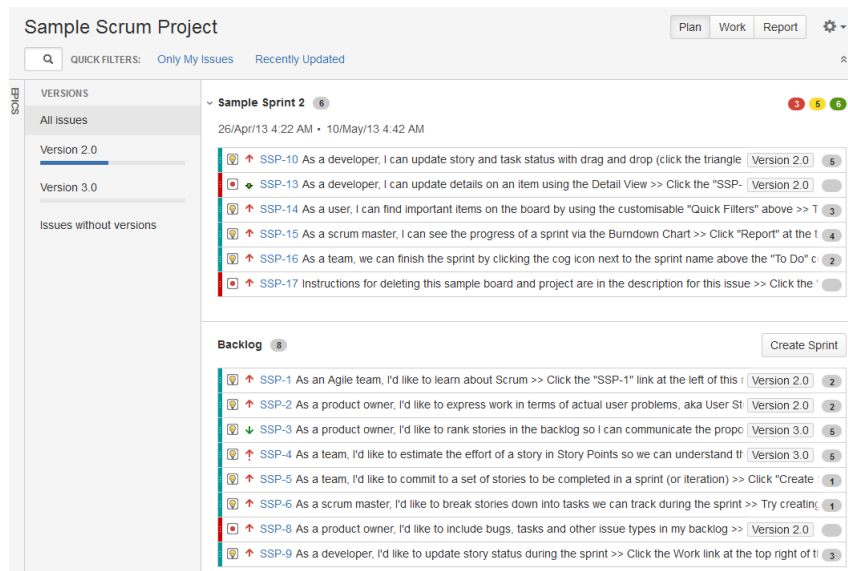


Fig. 1: A product backlog in Jira [Atlassian 2017]

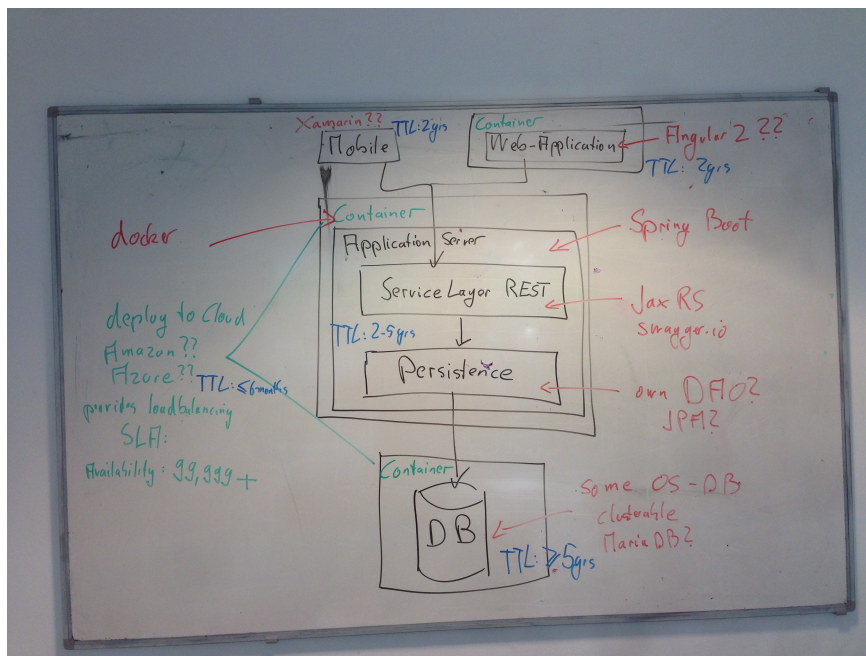


Fig. 2: Whiteboard drawing of architecture vision

*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.

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.

**3.4.2 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 YAML. The second example concerns tests written with Cucumber<sup>7</sup>. Both specifications are stored in a git repository.

Swagger.io<sup>8</sup> 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 shows an example of swagger in use. So the code itself serves as specification.

<sup>7</sup><https://cucumber.io/>

<sup>8</sup><http://swagger.io/>

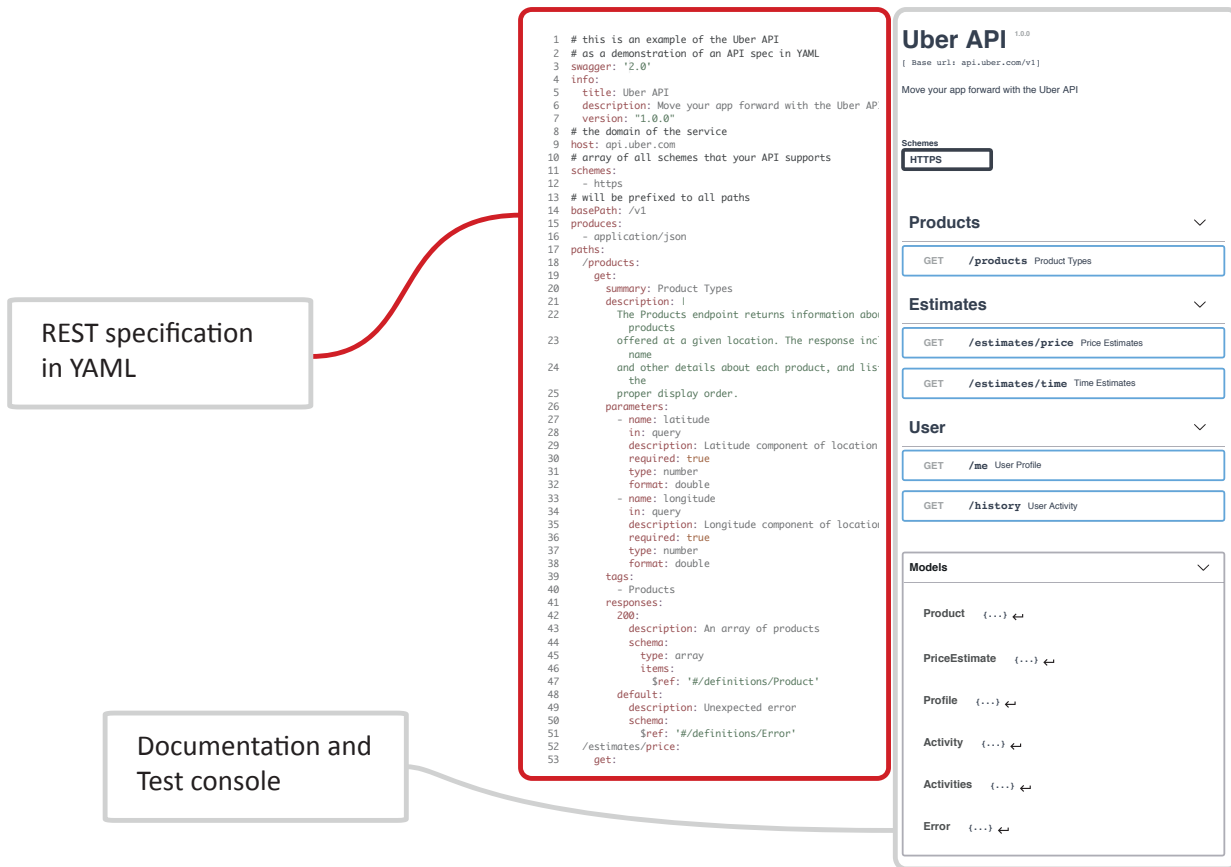


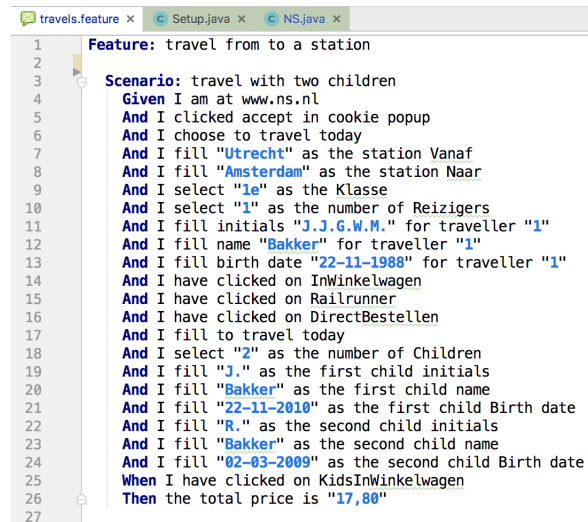
Fig. 3: REST API definition with swagger.io

Another example for a tool using the same principle is Cucumber. Cucumber is a behavior-driven development (BDD) [North et al. 2006] test tool. BDD was preceded by test-driven development (TDD) [Beck 2003]. In these types of development, tests are the specifications for the development team.<sup>9</sup> Cucumber is used for behavior-driven development in automated acceptance tests. Typically, a team with customers, developers and testers explore the area, each from its 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 4.

The principles also apply for many other types of artifacts generated using specific tools (e.g. UML-tools, 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

<sup>9</sup>We focus on the test as specification and do not present merits and (dis)advantages of test-first development methods.



```

1 Feature: travel from to a station
2
3 Scenario: travel with two children
4   Given I am at www.ns.nl
5   And I clicked accept in cookie popup
6   And I choose to travel today
7   And I fill "Utrecht" as the station Vanaf
8   And I fill "Amsterdam" as the station Naar
9   And I select "1e" as the Klasse
10  And I select "1" as the number of Reizigers
11  And I fill initials "J.J.G.W.M." for traveller "1"
12  And I fill name "Bakker" for traveller "1"
13  And I fill birth date "22-11-1988" for traveller "1"
14  And I have clicked on InWinkelwagen
15  And I have clicked on Railrunner
16  And I have clicked on DirectBestellen
17  And I fill to travel today
18  And I select "2" as the number of Children
19  And I fill "J." as the first child initials
20  And I fill "Bakker" as the first child name
21  And I fill "22-11-2010" as the first child Birth date
22  And I fill "R." as the second child initials
23  And I fill "Bakker" as the second child name
24  And I fill "02-03-2009" as the second child Birth date
25  When I have clicked on KidsInWinkelwagen
26  Then the total price is "17,80"
27

```

Fig. 4: Feature description in Cucumber for a Dutch train travel website

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, 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 [van Heesch et al. 2012a; 2012b].

Figure 5 shows an example of a decision-relationship-view [van Heesch et al. 2012a], which shows relationships between decisions made and their status. A relationship view in combination with a decision-forces view [van Heesch et al. 2012b]. Figure 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 7 shows an example of such an overview page created in Confluence.

**3.4.3 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

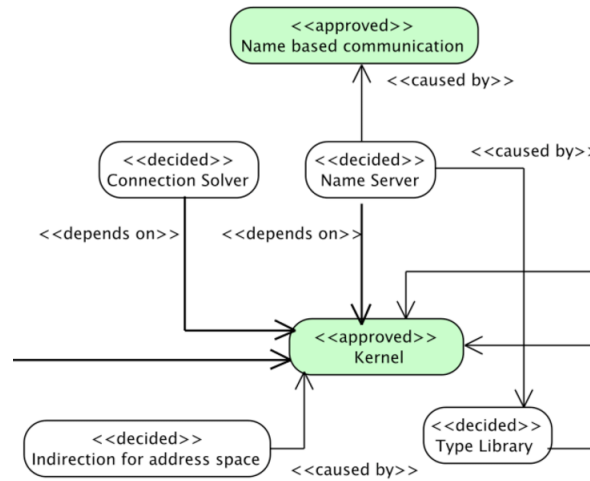


Fig. 5: Example of a decision relationship view

			View technology			Data storage	Middleware	DBMS	
			<decided>	<discarded>	<discarded>	<decided>	<discarded>	<discarded>	<decided>
			Java Swing	PHP	JSF	Central DS	EJB	MySQL	PostgreSQL
<b>Architecture significant requirements</b>									
Code	Description	Concerns							
R1	Avg. response time <= 0.1s	Time behavior	++	+	-	-	-	+	+
R5	Integrate multi. 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 3 <sup>rd</sup> party licences	Development costs	+	+	+			++	++
<b>Other forces</b>									
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	++	-	--	--	--	+	?

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 [Fairbanks 2010].

## Continuous Specs Sprint 3

Created by [REDACTED]

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.\[REDACTED\].nl/jenkins/job/ASD%20A%20Passenger%20Service%20-%20Develop](http://ci.[REDACTED].nl/jenkins/job/ASD%20A%20Passenger%20Service%20-%20Develop)
- Stash develop branch: [http://git.\[REDACTED\].nl/stash/projects/ASD1617S1A/repos/passenger\\_service/browse](http://git.[REDACTED].nl/stash/projects/ASD1617S1A/repos/passenger_service/browse)
- Sonar: [http://ci.\[REDACTED\].nl/overview?id=13561](http://ci.[REDACTED].nl/overview?id=13561)

### Technical Specs

- Restful API Specs: [http://ci.\[REDACTED\].nl/swagger/project/ASD%20A%20Passenger%20Service%20-%20Develop](http://ci.[REDACTED].nl/swagger/project/ASD%20A%20Passenger%20Service%20-%20Develop)
- Relational Schema Customer\_DB: [http://mysql.\[REDACTED\].nl/schemeGenerator?id=13561](http://mysql.[REDACTED].nl/schemeGenerator?id=13561)
- Virtual Machines and Container configuration: [http://proxmo:\[REDACTED\].nl](http://proxmo:[REDACTED].nl)

### Task and Issue Tracking

- Scrumboard: [http://jira.\[REDACTED\].nl/secure/RapidBoard.jspa?rapidView=394&view=planning.nodetail](http://jira.[REDACTED].nl/secure/RapidBoard.jspa?rapidView=394&view=planning.nodetail)

### Decision Views

- Decision Relationship View: [http://decision:\[REDACTED\].nl/relView?id=1761](http://decision:[REDACTED].nl/relView?id=1761)
- Decision Forces View: [http://decisions.\[REDACTED\].nl/forcesView?id=1761](http://decisions.[REDACTED].nl/forcesView?id=1761)


 Like Be the first to like this

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 et al. 2012a]. 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 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.

### 3.5 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 envi-

sioned 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.
- (4) **Overestimated competences and underestimated complexity** As a consequence of no big up-front 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) **Onboarding 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.

#### 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 [Kruchten et al. 2006] 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.

## 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.

## REFERENCES

- Scrum Alliance. 2017. What is Scrum? An Agile Framework for Completing Complex Projects-Scrum Alliance. (2017). <https://www.scrumalliance.org/> (Retrieved April 21, 2017).
- Atlassian. 2017. Visualise your Roadmap. <https://www.atlassian.com/blog/archives/visualize-your-roadmap>. (2017).
- Kent Beck. 1999. Extreme programming explained: embrace change. (1999).
- Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.
- Frederick P Brooks Jr. 1995. The mythical man-month (anniversary ed.). (1995).
- Daniel Cukier. 2013. 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, 143–152.
- Floris Erich, Chintan Amrit, and Maya Daneva. 2014. Report: Devops literature review. (2014).
- George Fairbanks. 2010. *Just enough software architecture: a risk-driven approach*. Marshall & Brainerd.
- Martin Fowler and Jim Highsmith. 2001. The agile manifesto. (2001).
- HP. 2016. Measuring DevOps success. How do you know DevOps is working? Watch the KPIs. <https://www.hpe.com/h20195/v2/GetPDF.aspx/4AA6-3036ENN.pdf>. (August 2016). (Retrieved April 5, 2017).
- Jez Humble and Joanne Molesky. 2011. Why enterprises must adopt DevOps to enable continuous delivery. *Cutter IT Journal* 24, 8 (2011), 6.
- Philippe Kruchten, Patricia Lago, and Hans van Vliet. 2006. Building Up and Reasoning About Architectural Knowledge. In *Quality of Software Architectures*, Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner (Eds.). Lecture Notes in Computer Science, Vol. 4214. Springer Publishing Company, 43–58.
- Justin Kruger and David Dunning. 1999. Unskilled and unaware of it: how difficulties in recognizing one’s own incompetence lead to inflated self-assessments. *Journal of personality and social psychology* 77, 6 (1999), 1121.
- Christopher Lovelock and Evert Gummeson. 2004. Whither services marketing? In search of a new paradigm and fresh perspectives. *Journal of service research* 7, 1 (2004), 20–41.
- Dan North and others. 2006. Introducing BDD. *Better Software, March* (2006).
- Mary Poppendieck and Tom Poppendieck. 2003. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc.
- Theo Theunissen and Uwe van Heesch. 2016. The Disappearance of Technical Specifications in Web and Mobile Applications: A Survey Among Professionals. In *Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28–December 2, 2016, Proceedings 10*. Springer, 265–273.
- U van Heesch, P Avgeriou, and R Hilliard. 2012a. A documentation framework for architecture decisions. *Journal of Systems and Software* 85, 4 (2012), 795–820.
- U van Heesch, P Avgeriou, and R Hilliard. 2012b. Forces on Architecture Decisions - A Viewpoint. In *Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture*. IEEE, xx—xx.
- Mandi Walls. 2013. Building a DevOps Culture. (2013).
- R Wilsenach. 2016. DevOps Culture. *Saatavissa (viitattu 23.4. 2016): http://martinfowler.com/bliki/DevOpsCulture.html* (2016).



# Appendices

## A. PRINCIPLES OF CONTINUOUS SOFTWARE DEVELOPMENT AND LEAN, AGILE AND DEVOPS

Principle of ConSD	Lean	Agile	DevOps
<p><b>1. Efficiency, effectiveness</b> One strives for an optimal balance between efficiency and effectiveness.</p>	<p><b>1. Eliminating waste</b> 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.</p>	<p><b>1. Customer satisfaction</b> <b>7. Working software is progress</b> Both customer satisfaction and working software relate to results for the customer. This includes short iterations for faster time-to-market and reducing risks.</p> <p><b>8. Sustainable pace</b> 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.</p> <p><b>10. Simplify</b> Simplification relates to efficiency by focussing on a) a minimum viable product, not more; 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 of 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, proven, easy to understand).</p>	<p><b>3. Measurement</b> 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 directed process improvements. On the other hand, these improvements can be evaluated by analyzing their impact on the respective measurements in retrospect.</p>

Principle of ConSD	Lean	Agile	DevOps
<p><b>2. Learning, improvement</b> Progressive insights.</p>	<p><b>2. Amplify Learning</b> Development is rather a creative process than a systematic process.</p>	<p><b>12. Regular adjustments</b> Improving team performance by reflections and regular feedback sessions.</p>	<p><b>4. Sharing</b> Sharing of knowledge, tools and successes.</p>
<p><b>3. Flexibility</b> Ability to adapt to new, unforeseen, and possible trivial situations.</p>	<p><b>3. Decide as late as possible</b> 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.</p>	<p><b>2. Welcome changes</b> The team welcomes changes to give the customer a competitive advantage.</p>	<p><b>7. Structures and standards</b> These are the <i>de facto</i> and <i>de jure</i> values, standards and behaviors within an organization that contribute to last minute changes. <i>De jure</i>: what is agreed upon, either by law, code of conduct or agreements. <i>De facto</i>: 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").</p>
<p><b>4. Time to market</b> The lead time it takes from concept to a minimal marketable product.</p>	<p><b>4. Deliver as fast as possible</b> Customers like fast delivery. Therefore, Lean strives for frequent and fast delivery.</p>	<p><b>3. Frequent releases</b> This includes minimum viable product (MVP) as well as minimum marketable product (MMP). Agile processes strive for delivering software increments after each iteration.</p>	<p><b>2. Automation</b> 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.</p>
<p><b>5. Trust, attitude</b> All parties trust and live up to the given trust.</p>	<p><b>5. Empower the team</b> Decisions are made inside the team and not imposed on the team.</p>	<p><b>5. Trusted individuals</b> Support the team with trust.</p>	<p><b>1. Culture</b> The culture of trust can be found in the whole of <i>de facto</i> and <i>de jure</i> values, standards and behaviors within our organization.</p>

Principle of ConSD	Lean	Agile	DevOps
<p><b>6. Competences</b> Highly skilled people who are experienced in a wide range of technologies.</p>	<p><i>This principle is compatible with the Lean mindset, but not explicit in the Lean philosophy.</i></p>	<p><b>9. Technical excellence</b> The development team employs technical skills, as well as process-related skills.</p> <p><b>11. Self-organizing teams</b> The competent team has clear objectives of <i>what</i> to achieve but limited rules on <i>how</i> to achieve these objectives.</p>	<p><b>6. Quality assurance</b> Defining characteristic that refers to the desired outcome, 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 the standards.</p>
<p><b>7. Big picture</b> The risk that people tend to excel in a specific skill while at the same time losing sight of the big picture.</p>	<p><b>7. See the whole</b> The risk of not having an overview, is that it may create an environment where suboptimal behaviour occurs with suboptimal results. Seeing the whole, or big picture, reduces suboptimal solutions as the big picture embraces the individual suboptimal solutions, internal competitions or.</p>	<p><b>6. Face-to-face conversation</b> 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 the other developers and stakeholders is mitigated by a continuous dialogue between developers and stakeholders.</p>	<p><b>5. Services</b> Teams focus on delivering added value, while leaving commodity solutions and non-core competences to others, e.g. service providers.</p>
<p><b>8. Involvement</b> Shared principles and priorities, understanding of one's concerns and standards.</p>	<p><i>This principle is compatible with the Lean mindset, but not explicit in the Lean philosophy.</i></p>	<p><b>4. Collaborate with business people</b> Understand each other's needs, possibilities and weaknesses.</p>	<p><b>7. Structures and standards</b> These are the <i>de facto</i> and <i>de jure</i> values, standards and behaviors within an organization, like e.g. a code of conduct. The structure and standards within an organisation encourage involvement , e.g. by training, knowledge-meetings, or time to experiment (like Google's<sup>1</sup> and Atlassian's<sup>2</sup> 20% rule).</p>

<sup>1</sup> <https://abc.xyz/investor/founders-letters/2004/ipo-letter.html>

<sup>2</sup> [https://www.atlassian.com/blog/archives/20\\_time\\_experiment](https://www.atlassian.com/blog/archives/20_time_experiment)