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

U. VAN HEESCH, T. THEUNISSEN, HAN University of Applied Sciences Arnhem, The Netherlands
O. ZIMMERMANN, University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland (HSR FHO)
U. ZDUN, University of Vienna, Vienna, Austria

We have been observing an ongoing trend in the software engineering domain towards development practices that rely heavily on verbal communication and small, closely-interacting teams. Among others, approaches like Scrum, Lean Software Development, and DevOps fall under this category. We refer to such development practices as Continuous Software Development (ConSD). Some core principles of ConSD are working in short iterations with frequent delivery, striving for an optimal balance between effectiveness and efficiency, and amplify learning in the development team. In such a context, many traditional patterns of software specification, documentation and knowledge preservation are not applicable anymore.

To explore relevant topics, opinions, challenges and chances around specification, documentation and knowledge preservation in ConSD, we conducted a workshop at the 22nd European Conference on Pattern Languages of Programs (EuroPLoP), held in Germany in July 2017. The workshop participants came from the industry and academia.

In this report, we present the results of the workshop. Among others, we elaborate on the difference between specification and documentation, the special role of architecture in ConSD in general, and architecture decision documentation in particular, and the importance of tooling that combines aspects of development, project management, and quality assurance. Furthermore, we describe typical issues with documentation and identify means to efficiently and effectively organize specification and documentation tasks in ConSD.

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

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

ACM Reference Format:

U. van Heesch, et al. 2017. Software specification in continuous software development - A Focus Group Report. *EuroPLoP* (July 2017), 13 pages.

DOI: https://doi.org/10.1145/3147704.3147742

1. INTRODUCTION

In the last decade, we have been observing a shift in the software industry towards software development practices that rely on verbal communication, closely interacting small teams, shorter planning and controlling horizons (often called sprints or iterations), and frequent delivery. Popular approaches include Scrum [Alliance 2017], Lean Software Development [Poppendieck and Poppendieck 2003], and more recently DevOps [Wilsenach 2016]. We refer to collections of these practices as *Continuous Soft-*

978-1-4503-4848-5/2017/07...$15.00
DOI: https://doi.org/10.1145/3147704.3147742

ware Development (ConSD) [Theunissen and van Heesch 2017] to take into account their predominant characteristics of continuous, short and time-boxed iterations and incremental refinement.

In continuous software development, some traditional patterns of software specification, documentation and knowledge preservation are not applicable anymore. One of the main challenges is dealing with software specifications and documentation in a continuous development flow. Specifications take many forms, e.g. requirements specification, architecture specification, design specification, test specification, and deployment descriptors. Team members primarily value specifications that have an immediate benefit for their own tasks during an iteration (e.g. specifications of interfaces between subsystems). However, depending on the type of software and the type of specification, these specifications may need to serve additional needs. For instance, they may need to support reasoning about technical risks, help stakeholders understand complex systems, support (offline) communication between stakeholders, or capture design decisions with long-term impact.

In our ongoing work on continuous software development (see for instance [Theunissen and van Heesch 2017]), we conjecture that the diverse types of specifications and documentation with different lifespans, different levels of formalism, different levels of detail, and different forms of codification should be handled individually and differently so that they can satisfy the diverse needs of stakeholders in ConSD. To further explore relevant topics, opinions, challenges and chances around specifications and documentation in ConSD (with a focus on architectural specifications), we organised a workshop at the 22nd European Conference on Pattern Languages of Programs (EuroPLoP), held in Irsee, Germany in July 2017. In this paper, we report on the results of the focus group and describe directions for future work on this topic.

The rest of this paper is organized as follows: Section 2 explains the setup of the focus group and the characteristics of the participants. In Section 3, we present the results of the discussion. Finally, Section 4 presents directions for future work on this topic.

2. WORKSHOP SETUP AND PARTICIPANTS

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

(1) What is the difference between specification and documentation? What purposes do they serve and for whom?

(2) What is the role of architecture specification in continuous software development?

(3) Should software specifications be organized around self-contained documents?

(4) What is the role of models in this context?

(5) What alternate forms of organizing specification items could be a better fit for continuous software development?

(6) What is the life cycle and scope of the different types of specifications and how does this life cycle relate to the agile life cycle, for instance?

(7) Refactoring has become a common technique for improving the structure and quality of source-code. How can similar techniques be used for specifications?

(8) What is the role of (architectural) design decisions in this context? How to share, document, or update them?

(9) How to efficiently preserve (architectural) knowledge and skills a development team gained throughout multiple iterations/projects? What difference exists between generic and application-specific knowledge and skills?
(10) How can agile practices (e.g. sprint retrospectives) be leveraged in this context?
(11) How can information needs by external reviewers and auditors be satisfied?
(12) What is the impact of business and technical domain specifics in this context?

Additionally, we presented a poster summarising our previous work on specification in ConSD (see Appendix A).

2.1 Participants

In total, we had 16 participants partly from industry and partly from academia. To find out more about the background of our participants, we asked them to fill in a questionnaire in the beginning of the focus group. The questionnaire and the results are available online[1]. As Figure 1 shows, roughly

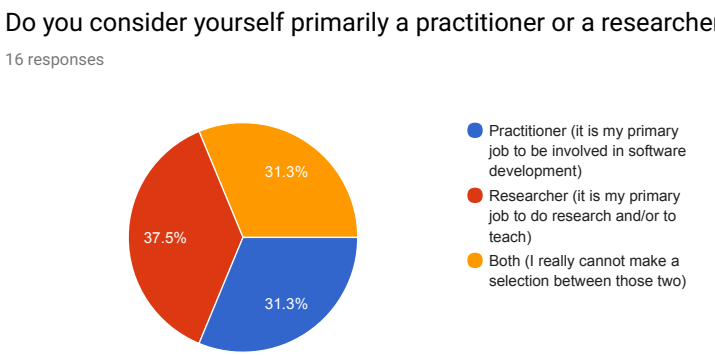


Fig. 1. Affiliation of participants

one third of the participants consider themselves as practitioners, the other participants are either academics or related to both industry and academia. We also asked the participants about the number of years in software engineering experience (see Figure 2). With an average of 13 years of experience, we mainly had senior software engineers in the workshop. Figure 3 shows the tasks, our participants are usually involved in when doing software projects. Most participants cover a wide range of typical software engineering tasks.

2.2 Setup

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

In the following, we summarize the findings of the groups.

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

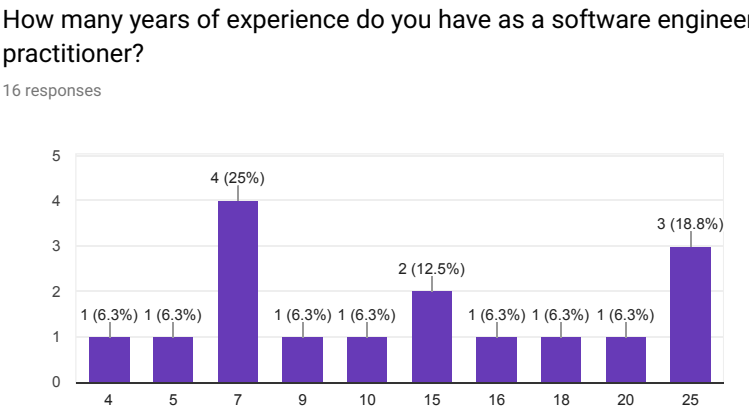


Fig. 2. Years of Software Engineering (SWE) experience

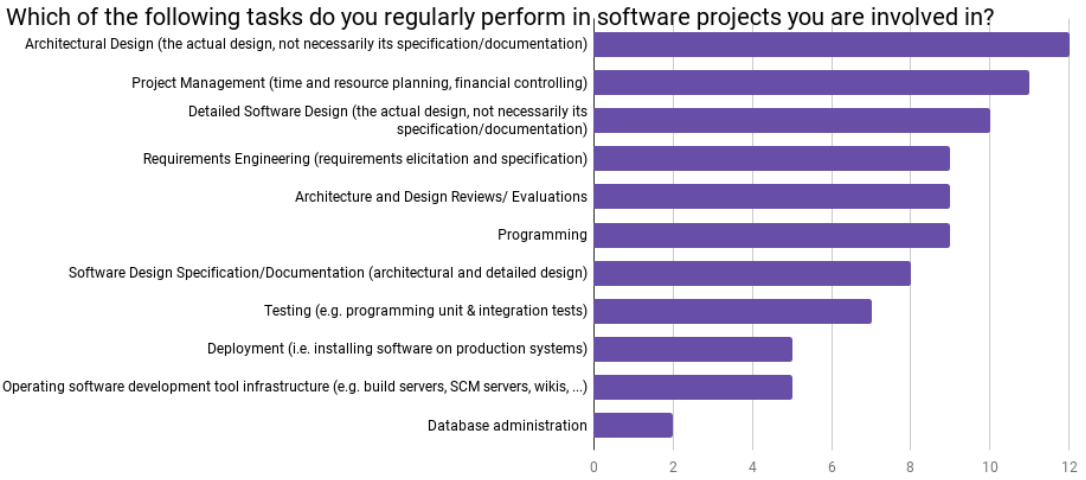


Fig. 3. Involvement in SWE tasks

3. RESULTS

We discussed the difference between specification and documentation; terms which are often used interchangeably in the software engineering domain. Documentation, as understood by our participants, is a piece of writing conveying information about a software artifact, to be consumed *after* the software artifact was built. The primary purpose of documentation is to help stakeholders better understand certain aspects of the system, be it because they need to support or further develop the system, approve certain aspects, review or audit it – or because they pay for it.

Specifications are seen as presentations of information meant to support the development process of a software artifact. Specifications are created and consumed *before or during* the development of a software artifact.

In many cases, specification artifacts are also used as documentation. These cases can be particularly problematic in ConSD, because specification artifacts are only maintained while they are necessary (or at least immediately beneficial) for the realisation process of a software artifact. As a consequence, the

actual realisation regularly derives from the last maintained state of the specification. When being used as-is as documentation, these artifacts contain inconsistencies and specification gaps and are thus less usable for the purpose of documentation.

In the following sections, we present additional insights about documentation and specification gained during the focus group.

3.1 Specification

Specifications are created upfront and give instructions, rules and guidelines on how software artifacts should be built or what properties they should have. They are usually prescriptive. Specifications are described as "living artifacts", as they are continuously adjusted with progressive insights, as long as they have a benefit for the development process. Different types of specifications exist, for instance requirements specifications, test cases, user interface specifications, architectural specifications, or design specifications (e.g. UML diagrams).

3.1.1 *Levels of formalism and detail.* We found that the degrees of formalism and completeness vary greatly between projects and artifacts. In ConSD, team members work together intensively, know each other well, and rely on oral communications. Therefore, informal rich pictures (e.g., white board sketches) are preferred over formal specifications. Formalisms are only used when being required by tools used (e.g. test specifications in Cucumber[2]), or because very detailed interface specifications are required (e.g., between a hardware team and a software team or between the provider and the consumers of a public API) [Zimmermann et al. 2017]). Related to this, teams develop a common body of knowledge, which consists of knowledge about specific technologies, patterns, and solutions applied in the past. Teams explicitly or tacitly refer to this common body of knowledge when creating specifications by leaving out details that external consumers of the specifications would require to form a complete picture, or by using a vocabulary that can be understood by the team members only. An example of the latter is the use of the term "request controller" in a specification, which if seen out-of-context, is so generic that it is meaningless for external people, while members of the team know exactly which component in an existing system is meant by the term.

Thus, teams who do not know each other well and do not share a common body of application-specific and application-generic knowledge, require more formal and more detailed specifications than teams who share such a body of knowledge. Experienced teams can apply a kind of specification-by-exception approach, in which only derivations from their standard way are specified. One participant stated it like this: "We know what we are doing, so no need to write everything down.". This phenomenon, in our perception, is independent of the type of specification.

3.1.2 *Architecture specification.* One break-out group explicitly discussed the role of architecture specification in ConSD. At first, the idea of creating comprehensive up-front specifications may seem to conflict with the principles of ConSD, such as avoiding waste and being agile. Nevertheless, the members of the focus group emphasised the important role of architecture reasoning and architecture specification at the beginning of larger projects or re-engineering efforts. Apart from the often-cited advantages of architecture like identifying risks and reasoning about quality attributes, architecture (primarily smart system partitioning) is seen as an enabler for agile working. This is pre-dominantly the case for larger systems which are too complex to be handled in the working memory of a human being at the same time. Smaller systems, and/or systems built the same way as other systems in the past, do not require much architecture specification. One participant said: "Let's not create large systems!" to express that systems that do not require architecture specifications to be manageable, are

[2]https://cucumber.io/

a much better fit for ConSD. Architecture issues arise mainly in large systems. If large systems are required by the nature of the problem, then architectural styles that split the system into small parts that can be developed and comprehended in isolation can be a solution. Microservices are one example of such an approach [Zimmermann 2017].

In ConSD, architecture specification is often done using a whiteboard and primarily describes the system partitioning into sub-systems and major components. For each component, the responsibilities and interfaces are described. Architectural design is discussed during iteration planning meetings (e.g. a sprint planning meeting in Scrum) and if required in a special "architectural stand-up" meeting during iterations.

3.1.3 *Tooling.* Especially in ConSD, teams rely heavily on integrated tool suites that combine aspects of project management with features of development tools. An example of such a stack is the combination of an IDE (e.g. IntelliJ[3] for Java), Git[4], JIRA[5] , Confluence[6], Jenkins[7], and SonarQube[8].

With such a tool stack, specification activities, coding, testing, quality management, and project controlling are closely intertwined and enable lightweight traceability between the different activities. To give an example, imagine the following typical flow (using tools mentioned by the participants):

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

The process sketched above shows how development activities are closely accompanied and supported by the tool chain. As a side effect, the tool chain provides traceability between requirements,

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

tasks, design artifacts, code, required time, and code quality. Apart from supporting the realisation of software artifacts (here a functional requirement), the information in the tools also serve documentation needs, as one can easily click through the history of executed tasks. The process above entails the following forms of specification:

—A requirements specification in form of a user story.

—Acceptance criteria, which likewise serve as acceptance tests.

—An effort estimation for the user story.

—A design specification in form of a white board sketch and additional agreements between developers specified in Confluence.

—Test specifications in the source code.

—Coding standards to be used in the IDE and in the CI-server.

—Further code quality standards to be used by SonarQube; some of these standards are architectural (e.g. regarding reliability, security, maintainability, and complexity).

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

While they are key artifacts for systems built in a model-driven way [Soley 2000], architecture specifications still seem to play a tangential role at least in some ConSD communities. That said, code quality tools support some forms of architectural analysis, as mentioned above. However, some popular examples exist of tools that use architectural specifications in the same way as the specifications described above. Ansible [9], for instance, is a tool for automating deployment tasks using so-called playbooks: text-based specifications of how production, staging, test, and development machines are set-up. Apart from automating deployment tasks, Ansible can be used for checking standard compliance, e.g., regarding security measures.

3.2 Documentation

As described above, we define documentation as artifacts meant to be consumed after a software artifact was realised. Especially in ConSD, the need and the effort spent on documentation is a sensitive topic, because developers strive for avoiding efforts that do not provide immediate value. Additionally, it can be stated that the vast majority of software developers does not like creating documentation [Herbsleb and Moitra 2001].

The focus group discussions related to documentation dealt with the following questions:

(1) For which purpose do we create documentation and what types of documentation do we observe in ConSD?

(2) When should we create documentation and how much is enough?

(3) What is the role of architecture decision documentation in ConSD?

(4) What problems can be observed regarding documentation?

(5) How to effectively and efficiently provide documentation in ConSD?

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

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

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

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

To support the automatical generation of design documentation, the participants in different breakout groups mentioned the tool Doxygen[10]. Doxygen is an example for a tool that creates documentation from source code and special annotations used in source code. Apart from providing textual explanations on modules, methods, parameters and the like, Doxygen can generate several graphs and diagrams to explain the structure of the software in terms of modules and packages. This approach is compatible with the mindset of the software craftsmanship school of thought, who proclaim that the truth is only in the code [Martin 2008].

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

3.2.3 *Architecture decision documentation.* Architecture decision documentation is a special type of documentation. The literature advocates thorough documentation of architecture decisions, e.g. using decision templates (e.g. [Tyree and Akerman 2005; Zdun et al. 2013]) or dedicated decision views (e.g. [van Heesch et al. 2012; Nowak et al. 2010]). During the focus group, the participants agreed that some form of architecture decision documentation is needed. However, the detail level and required comprehensiveness of architecture decision documentation was discussed controversially. Some participants stated that only the outcome of decisions is documented, but not the rationale behind the decisions

[10] http://www.stack.nl/~dimitri/doxygen

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

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

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

Version maze. Often, documentation is not explicit about the version of the software it describes. It is then unclear to the reader whether the information provided is still up-to-date. Even worse, documentation is sometimes compiled of items that describe different versions. In such cases, the documentation drastically looses merit.

Cyclic references. Documentation items often consist of multiple items or documents referencing each other intensively. Sometimes, important aspects are not thoroughly described, because documentation items reference each other in a cyclic way without actually providing the information at some place.

Incompleteness. The participants experienced that documentation often contains “TODOs” and gaps, i.e. parts of the system, or aspects, that are not described. This could be related to the phenomenon that many people write documentation incrementally rather than iteratively. The writer starts with a high level of rigor and comprehensiveness, but eventually misses time or motivation to continue in the same fashion. As a result, the documentation becomes more and more inaccurate, or contains large gaps. When wikis are used, empty pages or page stubs with ”under construction” as only content are a symptom of this issue.

Copy/paste of code or raw specification:. Sometimes, documentation writers take over large large snippets of code or design artifacts that are too detailed to effectively serve as documentation. One participant gave an example “A large class diagram with hundreds of classes, methods and parameters thrown at the reader has no value”. Sometimes, production data ends up in external documentation by accident; usage of text that violates Netiquette or is not politically correct has also been reported and qualifies as a specification/documentation smell as well.

Presentation planet. Very often, documentation is prepared in form of a slide-based presentation. This can easily be explained by the fact that managers and other non-technical stakeholders appreciate short summaries with a language they understand. Using slide-based presentations as the only means to document comes with several downsides. Presentations are often way too abstract for many purposes (especially for developers and operators), the slides alone are ambiguous

and require the “voice-over” to be understood, and they typically do not contain references to the realized items they document. Info decks, as described by M. Fowler [11]can be a reasonable compromise.

Zombie specifications. Also known as dead documents. Specifications and documentation items might not have any readership and might not have been updated in a long time. You can tell from missing references to them in meetings and other specification/documentation items, as well as from access logs. Such items qualify as waste from a lean management point of view; they should either be updated and improved to meet an information need in a particular target audience, marked as ”stalled” and archived, or discarded.

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

Split to stick to single responsibility. The single responsibility principle is taken over from agile software development [Martin 2003]. Interpreted in the context of documentation, the principle states that every specification or documentation item should cover one specific aspect or part of the software that should entirely be encapsulated by this documentation item. Groups of stakeholder concerns addressed by viewpoints can be used to source these responsibilities.

Apply open/closed principle. When refactoring specifications, make sure they are open for extension, but closed for modifications. This principle is adapted from the corresponding principle used by the agile software development community [Martin 2003]. Applied to specifications, you need to make sure that adding additional information does not require (major) rework of the existing documentation. Semantic versioning[12] should be applied to specifications and documentation items just like for code.

Remove repetition.. Do not repeat yourself is a lean and agile tenet, so one should not provide the same piece of information in different documentation artifacts. Repetition results in increased efforts and higher risk of inconsistency when information is changed. The ”definition of done” for specifications and documentation should include a check whether the same things have already been said elsewhere.

Replace specification by realisation. Related to the previous item, once a specified software artifact was realised, refer to the realisation instead of providing the same information in a different way. To give examples, instead of describing object interaction using a UML-sequence diagram, let the code speak for itself. Contemporary IDEs have become so powerful that exposing source code in the IDE is preferred by many developers over reading sequence diagrams.

[11] see https://martinfowler.com/bliki/Infodeck.html
[12] see http://semver.org/

Document general structure instead of concrete realisation. Instead of repeating information that is also provided by the source code itself, describe the general structure of a software artifact or pick one example and explain how the example can be adapted to other cases, as well.

Throw specifications away. Some specifications have fully served their purpose when a software artifact was realised. These specifications can be thrown away in the sense that they can be deleted and only kept in the history of the knowledge management tool (e.g. Confluence as mentioned above).

Provide yellow pages. Instead of striving for one large self-contained and complete document, split up documentation items into smaller coherent chunks and provide overview pages with links to these smaller chunks. A documentation chunk can be text, source code, a diagram, a model, a whiteboard sketch or anything else that has value as documentation.

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

4. FUTURE WORK

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

The research results will be used to develop a lightweight architecture framework that embraces the principles of continuous software development.

ACKNOWLEDGMENTS

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

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).

J. D. Herbsleb and D. Moitra. 2001. Guest Editors' Introduction: Global Software Development. *IEEE Software* 18, 2 (2001), 16–20.

R. C. Martin. 2003. Agile Software Development: Principles, Patterns, and Practices. (2003).

R. C. Martin. 2008. Clean Code: A Handbook of Agile Software Craftsmanship. (2008).

M. Nowak, C. Pautasso, and O. Zimmermann. 2010. Architectural decision modeling with reuse: challenges and opportunities. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge, SHARK 2010, Cape Town, South Africa, May 2, 2010*. 13–20. DOI:http://dx.doi.org/10.1145/1833335.1833338

M. Poppendieck and T. Poppendieck. 2003. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc.

R. Soley. 2000. Model driven architecture. *OMG white paper* 308, 308 (2000), 5.

T. Theunissen and U. van Heesch. 2017. Specification in Continuous Software Development. In *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee*.

J. Tyree and A. Akerman. 2005. Architecture Decisions: Demystifying Architecture. *IEEE Software* 22, 2 (2005), 19–27.

U. van Heesch, P. Avgeriou, and R. Hilliard. 2012. A documentation framework for architecture decisions. *Journal of Systems and Software* 85, 4 (2012), 795–820.

R. Wilsenach. 2016. DevOps Culture. (2016). http://martinfowler.com/bliki/DevOpsCulture.html

U. Zdun, R. Capilla, H. Tran, and O. Zimmermann. 2013. Sustainable Architectural Design Decisions. *IEEE Software* 30, 6 (2013), 46–53.

O. Zimmermann. 2017. Microservices tenets. *Computer Science - R&D* 32, 3-4 (2017), 301–310. DOI:http://dx.doi.org/10.1007/s00450-016-0337-0

O. Zimmermann, M. Stocker, D. Lbke, and U. Zdun. 2017. Interface Representation Patterns - Crafting and Consuming Message-Based Remote APIs. In *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee*.

Appendices

A. POSTER

Specification in Continuous Software Development

PRINCIPLES OF CONTINUOUS SOFTWARE DEVELOPMENT

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

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

CONTEXT

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

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

PROBLEM

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

Forces are the factors of influence that should be taken into account when deciding on a solution for the problem at hand. Forces are often conflicting.

FORCES

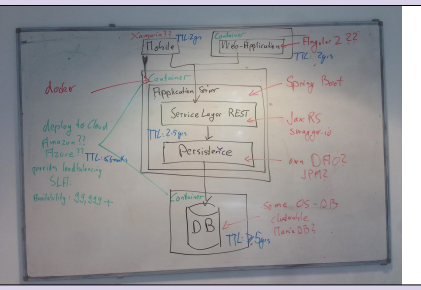
1. Shaping thoughts
   a. The process of specifying contributes to a better understanding of the problem and envisioned solution of an application.
2. Progressive insight
   a. During a software development process, developers gain new insights that they need to consider in the implementation
3. Specification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

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

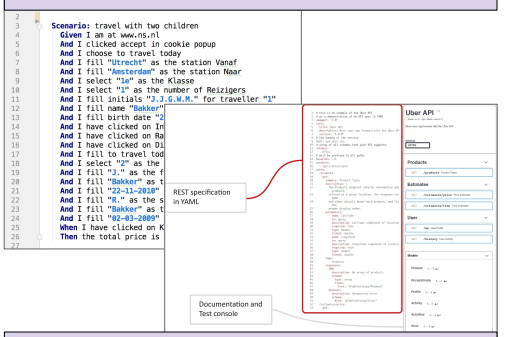
SOLUTION

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

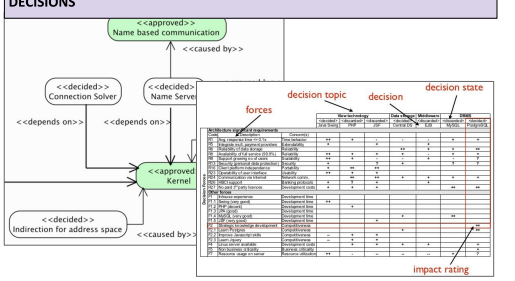
WHITEBOARD SKETCH



SPECIFICATIONS



DECISIONS



DISCUSSION ITEMS

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

Uwe van Heesch ● Theo Theunissen ● July 2017

Proceedings of the 22nd European Conference on Pattern Languages of Programs

europlop

Hogeschool van Arnhem en Nijmegen
HAN University of Applied Sciences