Towards an Anatomy of Software Craftsmanship

ANDERS SUNDELIN, Ericsson AB and Software Engineering Research Lab, Blekinge Institute of Technology, Sweden
JAVIER GONZALEZ-HUERTA, KRZYSZTOF WNUK, and TONY GORSCHEK, Software Engineering Research Lab, Blekinge Institute of Technology, Sweden

Context: The concept of software craftsmanship has early roots in computing, and in 2009, the Manifesto for Software Craftsmanship was formulated as a reaction to how the Agile methods were practiced and taught. But software craftsmanship has seldom been studied from a software engineering perspective.

Objective: The objective of this paper is to systematize an anatomy of software craftsmanship through literature studies and a longitudinal case study.

Method: We performed a snowballing literature review based on an initial set of nine papers, resulting in 18 papers and 11 books. We also performed a case study following seven years of software development of a product for the financial market, eliciting qualitative and quantitative results. We used thematic coding to synthesize the results into categories.

Results: The resulting anatomy is centered around four themes, containing 17 principles and 47 hierarchical practices connected to the principles. We present the identified practices based on the experiences gathered from the case study, triangulating with the literature results.

Conclusion: We provide our systematically derived anatomy of software craftsmanship with the goal of inspiring more research into the principles and practices of software craftsmanship and how these relate to other principles within software engineering in general.

CCS Concepts: • **Software and its engineering** → **Designing software**; *Software design techniques*; **Software development methods**; **Software development techniques**; **Collaboration in software development**; • **Applied computing** → *Electronic funds transfer*.

Additional Key Words and Phrases: software craftsmanship, principles of software development, deliberate practice

ACM Reference Format:
Anders Sundelin, Javier Gonzalez-Huerta, Krzysztof Wnuk, and Tony Gorschek. 2020. Towards an Anatomy of Software Craftsmanship. 1, 1 (May 2020), 50 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The notion that programmers should be responsible for what they produce has early roots. Already in 1975, Brooks [13] mention "invention and craftsmanship" as prerequisites for efficient optimization techniques, and he also envisioned "the surgical team" as an efficient way of developing mission-critical software. In 2002, McBreen published a book [57], formalizing the software

Authors' addresses: Anders Sundelin, anders.sundelin@ericsson.com,anders.sundelin@bth.se, Ericsson AB and Software Engineering Research Lab, Blekinge Institute of Technology, Karlskrona, Sweden; Javier Gonzalez-Huerta, javier.gonzalez.huerta@bth.se; Krzysztof Wnuk, krzysztof.wnuk@bth.se; Tony Gorschek, tony.gorschek@bth.se, Software Engineering Research Lab, Blekinge Institute of Technology, Karlskrona, Sweden.

XXXX-XXXX/2020/5-ART $15.00
https://doi.org/10.1145/1122445.1122456

craftsmanship concept, and since then, several books have been written on the subject [52, 54–56]. Another early inspirational work was published in 1999 by Hunt and Thomas [38].

The Manifesto for Software Craftsmanship[1] was published in March 2009, seven years after the Agile Manifesto[2]. The original signatories intended to address what they saw as deficiencies in how the Agile Manifesto principles had turned out in practice, as taught by coaches and certified institutions, and to emphasize the need to "make the thing right." The Software Craftsmanship movement lives on, twelve years after the manifesto was published. There are associated communities and conferences such as Socrates[3] in Europe and SCNA[4] in North America. However, we have not found any systematic definition of software craftsmanship principles and practices in research.

This paper moves towards this goal by providing an anatomy of software craftsmanship based on a systematic literature study and a longitudinal case study of a software product developed by an organization that was following software craftsmanship principles. In doing so, it moves towards systematizing and making explicit the software craftsmanship principles and practices to the broader research community, as there seems to be a lack of research papers in this area, as evidenced in Section 4.

The case study subject was a unit within Ericsson developing a new software product for seven years. The product operates in the financial sector and is in use in around twenty installations around the world. Due to the stringent requirements of financial systems and the values of the developing organization, the product was developed from scratch, highly inspired by craftsmanship principles, such as test-focused, agile, and lean software development, with a high focus on clean code and refactoring. These principles were also spread to new developers joining the product.

The paper is structured as follows: In Section 2, we give the background and related work of software craftsmanship and define the terms we use throughout the paper. In Section 3, we report on our research methodology, with Section 3.1 focusing on the systematic literature study, Section 3.2 focusing on the case study methodology, including the studied context, and Section 3.3 focusing on the process of building the anatomy. In Section 4, we report on the results of the SLR, and in Section 5, we merge this with the quantitative and qualitative results of the case study to produce our version of the anatomy of software craftsmanship. In Section 6, we discuss the implications for the software development community at large. In Section 7, we discuss the threats to the validity of the study. In Section 8, we draw on the analysis, outline future work and research directions, and make conclusions.

2 BACKGROUND AND RELATED WORK

The Craftsmanship movement builds upon Agile and Lean principles and practices, but with a stronger emphasis on building high-quality products by teams with a shared professional culture. The Manifesto for Software Craftsmanship was published in March 2009, following a summit in December 2008, where around 30 participants gathered to discuss what they perceived had been lost as the software industry adopted the Agile Manifesto. In particular, the lack of focus on the more technical practices in Agile processes such as Extreme Programming (XP) was a concern.

There have been several books and seminal works before 2008 (e.g., the books by Brooks [13], Hunt & Thomas [38], McBreen [57], Martin [54–56] and later also Mancuso [52]) that provide insights into the concept, the practices, and the potential benefits of Software Craftsmanship. However, very few research works delve into the formalization of the concept, with its principles

[1]http://manifesto.softwarecraftsmanship.org/
[2]http://www.agilemanifesto.org/
[3]https://www.socrates-conference.de/
[4]https://scna.softwarecraftsmanship.com/

and practices, with buttressing, real-world empirical evidence from cases where craftsmanship principles were put into operation.

If we look at the Agile Software Development, on the one hand, there are a plethora of Systematic Literature Reviews (e.g., [41, 74, 84]), Systematic Mapping Studies (e.g., [24]) and even Tertiary Studies (e.g., [36]) that portray how academia has studied Agile Software Development. In addition, several studies report on the benefits of Agile and XP practices in industrial settings (e.g., [42], [26], and [2]). Likewise, multiple studies address the potential benefits and drawbacks of Test-Driven Development, with several experiments (e.g., [30, 83]), case studies (e.g., [26]), and Systematic Literature Reviews (e.g., [61])

Lean Software Development was popularized by Poppendieck & Poppendieck [67] and has been studied in an industrial setting [65, 66]. Several Systematic Literature Reviews and Systematic Mapping Studies report results on metrics related to Agile and Lean software development and their relevance in the software industry [15, 27, 46].

3 RESEARCH METHODOLOGY

This paper uses a systematic literature review (SLR) method, using Wohlin's snowballing approach [90], and a case study method following guidelines by Runeson et al. [73]. We focus on the following research questions:

- **RQ1** How has prior literature described the principles and practices of software craftsmanship?
- **RQ2** Which of the identified principles and practices can we see applied in a real-life, commercial case study?
- **RQ3** What are the consequences of applying these principles and practices of software craftsmanship?

We aim at answering RQ1 by performing a systematic literature review. We aim at answering RQ2 by collecting quantitative measures on the studied system and triangulating them with interview findings with developers and the lead architect of the product. RQ3 is answered by extracting and synthesizing the literature review results and combining them with case study findings.

3.1 Systematic Literature Review Methodology and Execution

We conducted a systematic literature review using the snowballing method described by Wohlin [90]. We used a hybrid search strategy by combining the database search with iterative citations and references analysis [60]. Forward snowballing (citation analysis) greatly improves the precision, while backward snowballing (references analysis) greatly improves the recall of literature reviews.

3.1.1 Start Set Identification. We performed a database search in Google Scholar in December 2018, using the terms "software craft" OR "software craftsmanship" OR "software craftsman" OR "software craftsmen" OR "software craftsperson." We got 980 results that were analyzed by two authors, based on the following criteria:

(1) Is the paper published in an English-language journal, conference, or workshop proceedings, indexed by Google Scholar?
    This step excludes books, book reviews, and thesis works, including M.Sc. and Ph.D. theses.
(2) Does the paper describe themes, practices, or otherwise conceptualize software craftsmanship?
    This step excludes articles only referring to other works, such as [54], without providing any additional detail.

Criterion 1 excluded 522 papers and criterion 2 excluded 346, resulting in 112 papers, which were screened as potential seeds. Based on analysis of the title and abstract, we selected papers discussing various aspects of software craftsmanship, which resulted in four initial seed papers, denoted P1, P2, P3, and P4. According to Wohlin [90], the start-set should include papers from different publishers, authors, communities and should not be too small. Since diversity and scale are important for snowballing, we decided to broaden our set with relevant papers identified from our experience and recommendations, not only the database search. After some initial deliberation and analysis, we decided to add another five seed papers, denoted P5, P6, P7, P8, and P9. We also decided to drop our initial requirement to include only peer-reviewed papers since some of the included papers are magazines. At least two researchers applied the inclusion and exclusion criteria. When two reviewers had an initial disagreement, the conflicts were resolved by consensus.

3.1.2 Snowballing iterations. We performed four snowballing iterations summarized in Table 1 and stopped when we found no new relevant papers, applying the inclusion and exclusion criteria following the process described in Section 3.1.1. The full results of the SLR are available here[5].

Since the Software Craftsmanship concept comes both from the Craftsmanship Manifesto and seminal books, we extended the literature review with the final forward snowballing iteration focusing on books. In other words, we followed the references of the found papers and created a pool of books ready for analysis by partially following the guidelines for Multivocal Literature Reviews presented in [32]. This resulted in 146 books. As in the protocol we followed for "white" literature, two researchers applied the inclusion and exclusion criteria, and the conflicts were resolved by consensus. We divided the books between three of the researchers by letting each researcher analyse two-thirds of the books, making sure each book was reviewed twice. After applying the second exclusion criterion (2), we discarded a total of 135 books. The pairwise Cohen's Kappa results are 1.0, 0.59, and 0.48, which is less than the recommended criteria of 0.7. All three researchers discussed the seven books where disagreements were identified, and four of these were included in the final result after consensus had been reached. We decided not to iterate on other works citing included books since the number of citations for the included books is extremely high, and the main references from the paper-set had already been included. Section 4 contains the full results of the Systematic Literature Review.

Iteration	Number of citations and references screened	Included papers and books
Seed-1		P1 [81], P2 [62], P3 [71], P4 [50]
Seed-2		P5 [51], P6 [40], P7 [20], P8 [68], P9 [53]
Iteration 1	213 references and 186 citations	P10 [64]
Iteration 2	30 references and 1 citation	P11 [49], P12 [82], P13 [63], P14 [9]
Iteration 3	217 references and 517 citations	P15 [76], P16 [7], P17 [48]
Iteration 4	18 references and 78 citations	P18 [89]
Ref. Books	146 referenced books	B1 [13], B2 [57], B3 [47], B4 [54] B5 [78], B6 [37], B7 [77], B8 [55] B9 [52], B10 [35], B11 [88]

Table 1. Snowballing iteration statistics and results

3.2 Case Study Methodology

The goal of the case study is to analyze different craftsmanship practices followed in developing a product over seven years.

[5] https://tinyurl.com/Sundelin-SWC-SLR

3.2.1 The Case. The product studied in the case study is a FinTech global product that enables access to financial services via mobile phones and the Internet. The system is a high-availability, transaction-intensive product, with incoming and outgoing interfaces, a database, and scheduled tasks such as sending notifications. As it operates in the financial sector, security plays a central role in development.

Our investigation focuses on the financial core, containing the core business logic, such as financial transaction management, and associated user interfaces. A deployed product also contains other components (both third-party hardware and software) and customer adaptations, which are out of our analysis scope. All other components use the services of the core to perform financial tasks. The system is built in Java, using EJB 3[6] patterns, and uses a custom framework for deployment.

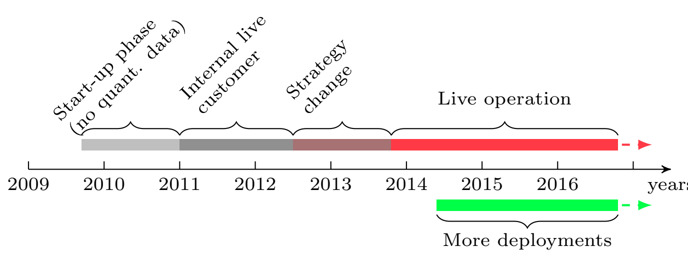

Fig. 1. Timeline of major events in the studied system.

Figure 1 depicts the timeline of the studied period, together with major events in the life cycle of the product. The first line of code was written in September 2009, and the first live demo for external parties was held in late October 2009. During 2011, Ericsson's strategy was to provide the solution as a service for end-users, and the system was deployed and taken into live operation in this manner. Following a business strategy change, the company decided to decommission the service and adopt a product-line approach. In late 2013 the first installation of the product went into operation at a customer site. Subsequently, the roll-outs continued, and the product was serving several tens of millions of end-users in more than 15 deployments worldwide during 2016.

As of December 2010, there are quantitative data available in the Git Version Control System. Before that, the project used ClearCase, a licensed product whose storage is unavailable for analysis.

The initial phase of the product (between 2009 and 2011) can be characterized as "the startup phase," with frequent changes of direction and no market deployment. Between 2011 and 2013, the internal customer provided feedback on the operation and deployment of the system. When the first external customer contract was signed in 2013, and the first system was taken live later that year, the direction became more stable, with increasing inflow of customer requirements.

The product used one primary and one supporting development site for most of the studied period. From mid-2011 until mid-2012, one development team was based in China. Following a change in product strategy, in mid-2013, two development teams from India were on-boarded instead, and this continued until the end of the studied period.

During the whole studied period, ending in December 2016, the product has been developed in an agile manner, first using two-week and later three-week sprints, heavily inspired by the craftsmanship principles and practices, as discussed in Section 5.

During the studied period, 155 individual developers have contributed to the studied system (measured via the Git *Author* tag). The first author of this paper was a developer from the project

[6]http://download.oracle.com/otndocs/jcp/ejb-3.1-pfd-oth-JSpec

start until October 2016. Table 2 contains the distribution of developers per quarter and quarters per developer. On average, 48.9 developers contributed to the code base each quarter. The peak of activity was reached in Q2 2016 with 91 contributors. In total, 24 quarters were studied, and in 75% of these, more than 36 authors contributed code. This clearly shows that the product is larger than what a single agile team can accommodate, requiring inter-team collaboration and communication.

Table 2. Descriptive statistics related to the number of developers in the system

Metric	$\bar{x}$	σ	$Q_{25\%}$	$Q_{50\%}$	$Q_{75\%}$	Min		Max	
Developers per quarter	48.9	17.7	36	48	53	25	(Q1 2011)	91	(Q2 2016)
Quarters per developer	7.6	6.5	3	5	11	1	(14 dev)	24	(5 dev)

On average, each developer stayed almost two years (7.6 quarters) in the product, though 50% of the total 155 authors contributed five quarters or less, and 25% contributed three quarters or less. This turnover data for the studied period show similar characteristics as the cases reported in previous research in the area [16]. The distribution is slightly right-skewed, as indicated by the minimum and maximum values, with five authors contributing during all 24 studied quarters and 14 authors contributing during a single quarter.

Although most contributors have been software developers, more persons and roles such as requirement engineers, system testers, product-, project- and line managers have contributed to the product. These roles are not studied in this paper.

3.2.2 Data Collection. We used two data collection methods. We gathered qualitative data through interviews with different roles involved in developing the product at different points. We also gathered data using archival analysis, using different artifacts (e.g., Version Control Systems, documentation, requirements, and defect reports) to measure the potential effects of craftsmanship practices on the product and the development process. We interviewed six participants for this case study, two female and four male subjects. Two of the interviewees worked in India, and four worked at the primary development site. Table 3 details the participants' background, as well as the legend used in citations and tables.

The interviews were organized as semi-structured interviews, using the interview instrument to structure the discussion. The interview protocol, which is publicly available here[7], was built and reviewed by the researchers and adapted as the interviews progressed to focus more on each interviewee's areas of expertise. At least two researchers conducted all the interviews, intervened in the discussion at will, clarifying statements, and introducing new topics and areas. All the interviews were audio-recorded and transcribed before analysis.

Table 3. Case study interviewee background, ordered by industry experience

Legend	Description	Experience
SwArch1	Lead Architect	20+ years in industry, 8 years in the product, starting 2009
Test2	Test-focused developer and Scrum master	≈20 years in industry, 8 years in the product, starting 2009
Test1	Test-focused developer and Scrum master	≈15 years in industry, 2 years in the product, starting 2015
Dev2	Developer	≈15 years in industry, 4 years in the product, starting 2013
Dev1	Developer	≈10 years in industry, 4 years in the product, starting 2013
Dev3	Developer and Scrum master	≈10 years in industry, 5 years in the product, starting 2012

[7] https://tinyurl.com/Sundelin-SWC-Interview

3.3 Consolidated Data Analysis: Building the Anatomy

In this subsection, we describe how we analyzed both the SLR and case study results.

The interview transcripts and the SLR results were analyzed using Thematic Analysis (TA), following the guidelines by Braun & Clarke [10]. We opted for TA since we were not exploring a completely alien phenomenon (i.e., Software Craftsmanship). Therefore there is no need to build an entirely new theory that emerges directly from the data, as is one of the main strengths of Grounded Theory [34], that in general is better suited to answer broader questions, such as *"what is going on there?"* [80]. TA is a robust and systematic framework for coding and analyzing qualitative data, identifying patterns across datasets in relation to research questions [11]. TA is also best suited when most of the collected data belong to a precise context, which then will move to generalizations and finally will allow building theories [3]. We carried out a theoretical or *deductive* approach for Thematic Analysis[10] by starting with a *theory* (a set of codes and themes), updating this as new data emerged.

Figure 2 summarizes the process for building the Anatomy of Software Craftsmanship. We first generated the initial set of codes (i.e., craftsmanship principles and practices), represented in the form of a mind-map (i.e., the Anatomy). This first set of codes was built based on the Software Craftsmanship Manifesto and themes from books, as indicated in Table 5. The first author then discussed the initial anatomy with the other authors in devil's-advocacy-type sessions.

Then the papers and the books included from the SLR were analyzed and coded, searching and reviewing the emerging codes and themes. When coding the books included as grey literature, two researchers read each book. Once the coding was finished, the two researchers met to discuss the codes found and went through the coding conflicts, which were solved by consensus.

The next step was coding the interview transcripts. The first author performed the initial In-Vivo coding [75] of all six interviews. Next, the second and third authors independently coded three transcripts each, assuring that at least two independent researchers coded each interview, prioritizing the interviews in which each researcher was present. Once coding was finished, the researchers met to discuss the potential coding conflicts, which were resolved by consensus. The coding was done using the corresponding version of the Anatomy with the codes. During the coding process, codes were merged, renamed, and new codes and themes were identified and added to the Anatomy, as suggested in Figure 2. This process triggered the need to review the already coded materials to identify potential instances of the new codes and themes in the data.

Taking the "Requirements" concept as an example to illustrate the process:

(1) The first author of this paper had experience from the case study, as well as noting the importance of localized customers, as stated in several of the reviewed books, see Table 5. Based on this, he initially decided on the code *On-site customer*, as it is a concept from Extreme Programming [5] (XP) that aligns well with the requirements process of the case study. After discussions with the additional authors, this code was used to explore the SLR results and to guide the interviews. However, neither the coding system nor the tentative map was shown to the interviewees before the interview.
(2) Both books B2 [57] and B8 [55] mention the importance of communication between development teams and requirement owners, indicating that the requirements concept should be somewhere near the Feedback theme.
(3) Furthermore, while conducting interviews, evidence was made more apparent that requirements were written in cooperation between the developers and the *On-site-customer* (though the case study used the Scrum term "Product Owner" (PO)). This was mentioned by several interviewees, for example, "We had our requirements in [the wiki-based requirement tool]. And the PO owned them — or the team — sometimes the team helped formulate them. But

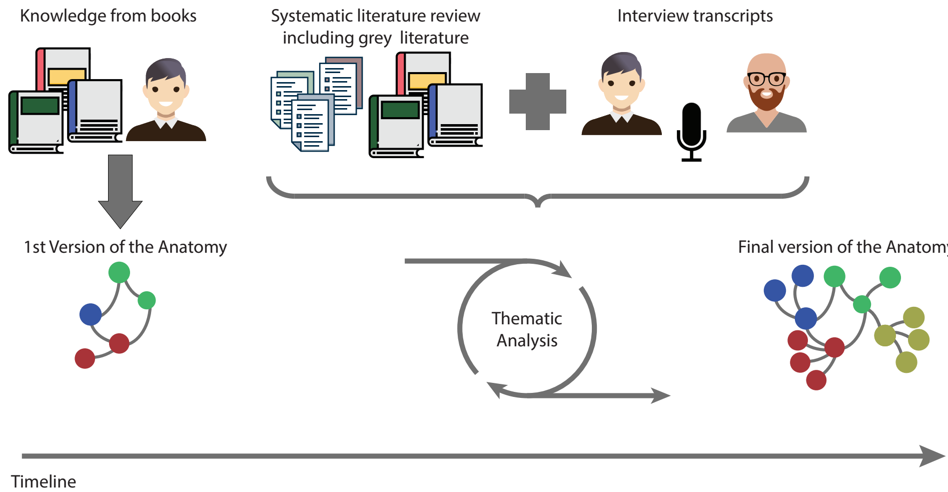

Fig. 2. Process for Building the Anatomy of Software Craftsmanship.

you walked through them [with the PO]. In [a different product], where I am now, it is completely different…"(Test2)

(4) Two other interviewees (Dev1, Test1) also indicated that the requirements were collaborative, mentioning the importance of looking "top-down" while simultaneously keeping a "bottom-up" perspective. This was also found in seven books and two papers in the literature, see Table 11 for details. In B2 [57], McBreen cites a study by Curtis, Krasner, and Iscoe, where this was stated as "Characteristically, customers also underwent a learning process as the project team explained the implications of their requirements. This learning process was a major source of requirements fluctuation." [23]

(5) The importance of *Accessible* requirements was also made clearer during the interviews. Having a clear, accessible requirement base was important for being able to work in parallel: "A strength in [the case study project] was that we could start testing in parallel with development. And we had clear requirements in one place [the wiki-based requirement tracking tool]. Based on this, the developers did their analysis, and testers did theirs in parallel. So we could write our acceptance test cases while development was ongoing."(Test2). Another interviewee supported this claim, and eventually, the *Accessible* requirement code was also found in book B2 [57] and papers P3 [71] and P9 [53].

(6) Based on these data points, we decided to add the **F1.1.2** *Collaborative* and **F1.1.1** *Accessible* practices to the **F1.1** *Requirements* practice, connected to the original *On-site-customer* principle. The decision to keep the whole sub-tree in the **F** *Feedback* theme was confirmed while analyzing additional data, such as when an interviewee discusses interactions between the requirements owner and the development team: "I would say we talk to [the requirements owner] every day, almost…Or, maybe at least for half an hour every other day…It's quite often we encounter things, in code and so on, that is not really how the requirement was imagined…Then you have to discuss that."(Test1). In total, four interviewees, three books, and three papers confirmed the importance of **F2** *Short feedback loops* between requirements engineers (regardless of title or term used), the development team, and the verification engineers.

(7) While this paper was in revision, a reviewer rightly pointed out that our so-called "On-site customers" were not really customers, but mere proxies for real, paying customers. Therefore, we decided to rename the principle to **F1** *On-site customer (proxies)*, indicating that sometimes you have to work with proxies for real customers (or end-users).

To increase validity and get feedback on our work, we shared the interview transcripts with the interviewees to ensure that we properly captured their opinions. We also presented an intermediate version of the craftsmanship map for company employees, including those currently working with the product. This provided valuable, though unstructured, feedback, which validated our structure.

We used statistical methods such as descriptive statistics and graphical representations to analyze and describe the case study's quantitative data.

4 SYSTEMATIC LITERATURE REVIEW RESULTS (RQ1)

In this section, we summarize the main findings of the Systematic Literature Review. The results of the consolidated thematic analysis will be outlined in Section 5. Table 4 outlines the results of the analysis of papers P1 to P18. Only 6 out of 18 papers can be considered empirical studies. Opinion papers and personal experience papers dominate the non-empirical studies and receive rigor scores between 0 and 1 and relevance scores between 1 and 2, making these papers partly relevant for our work. We used rigor and relevance criteria proposed by Ivarsson and Gorschek [39]. Rigor can have scores from 0 to 3 and is related to describing the context (maximum 1 point), study design (maximum 1 point), and validity (maximum 1 point). Relevance can have scores from 0 to 4, considering industrial participants (max 1 point), industrial context (maximum 1 point), realistic size of the study (maximum 1 point), and the usage of research methods that facilitates investigating real situations (maximum 1 point).

Among the non-empirical papers, two papers view craftsmanship from the perspective of the history of software engineering. Among them, P18 gives a brief history of Software Engineering, referring to Dijkstra declaring programming to be a discipline rather than a craft. Paper P14 also looks into the history of Software Engineering and uses the term "software crafting" to describe the (lack of stringent) processes for programmers during the 1960s.

On the philosophical stance, papers P11 and P17 discuss the theoretical underpinnings of the epistemology of craft in modern programming. Paper P1 provides a similar discussion, advocating that software methods should find ways of incorporating vernacularism and objects to a strictly rational software design process.

Six non-empirical papers present opinions, visions, or experiences. Among these, paper P6 argues that engineering is a craft supported by theory, while paper P16 argues that professional practice is craftwork. Paper P8 discusses the general craftsmanship model and the software craftsmanship model. Paper P7 highlights the importance of craftsmanship. Paper P9 focuses on the relation between agile and craftsmanship, and paper P12 brings opinions about using katas. Paper P15 summarizes experiences holding a course involving craftsmanship principles.

None of the six empirical papers takes a holistic view of software craftsmanship. Instead, they focus on practices (e.g., a community of practice for papers P3 and P10; craftsmanship forums and chats for paper P2; using katas to learn and improve for paper P13).

Empirical papers P4 and P5 are the closest to this work. Paper P4 empirically derives different conceptualizations of craft in building software, using a sample of 12 participants, whose subjective opinions were collected via interviews and a focus group. Paper P5 attempts to outline the craftsmanship practices based on the experiences from a project run with Scrum. The paper discusses steadily adding value vs. responding to change, a community of professionals, customer collaboration, and productive collaboration. Despite being highly relevant, paper P5 appears to

Table 4. Results from the systematic literature review

Paper [ref]	Found in	Refs	Cited	Rigor	Relevance	Venue	Year	Empirical	Main contribution
P1 [81]	Seed1		P10	0	0	Journal	2003	No: vision paper	Craft metaphor for software creation
P2 [62]	Seed1			4	3	Journal	2013	Yes: qualitative and quantitative, longitudinal study	Craftsmanship forums and chats as a part of community of practice
P3 [71]	Seed1			3	3	Conf.	2013	Yes: questionnaire and focus groups	Community of practice as a part of software craftsmanship
P4 [50]	Seed1			2	3	Conf.	2014	Yes: qualitative interviews and focus groups	Different conceptualizations of craft in building software
P5 [51]	Seed2			1	4	Workshop	2016	Yes: experience report	Analyzes software craftsmanship values in a Scrum project
P6 [40]	Seed2			0	2	Magazine	2014	No: opinion paper and anecdotal evidence	Engineering is craft supported by a theory
P7 [20]	Seed2			0	2	Non-academic conference	1994	No: experience report mostly based on anecdotal evidence	Stresses the importance of craftsmanship
P8 [68]	Seed2			0	2	Non-academic journal	2003	No: opinion paper	Discusses general craftsmanship and software craftsmanship models
P9 [53]	Seed2			1	2	Workshop	2008	No: personal experience	Focus more on agile than craftsmanship
P10 [64]	Iter1	P11, P12, P13, P14	P1	2	3	Journal	2015	Yes: qualitative and quantitative surveys	Community of practice and software design
P11 [49]	Iter2		P10, P16	1	2	Journal	2013	No: theoretical	Epistemology of craft in modern programming
P12 [82]	Iter2		P10	0	1	Non-academic journal	2010	No: opinion paper	Katas as a part of craftsmanship
P13 [63]	Iter2	P15	P10	2	2	Magazine	2014	Yes: experiment using katas	Katas as a way of learning and personal improvement
P14 [9]	Iter2		P10, P16, P17	0	1	Conf.	2006	No: opinion paper	The birth of the crafting paradigm preceding SE in the 1960s
P15 [76]	Iter3		P13	1	2	Conf.	2012	No: personal experience of the course instructor	Courses that involve craftsmanship practices
P16 [7]	Iter3	P11, P14, P17	P18	0	1	Conf.	2016	No: observations of the authors	Professional practice is craftswork
P17 [48]	Iter3	P14	P16	1	1	Workshop	2012	No: previous version of P11	Previous version of P11
P18 [89]	Iter4		P16	0	1	Journal	2008	No: personal opinion paper	Mentions craftsmanship in the history of SE

be an experience report from a project manager's point of view. The paper provides quantitative analysis of technical debt (number of lines removed over time) and velocity in backlog hours versus Sonar Cute estimated technical debt. However, the paper lacks systematic connection between the presented experiences and evidence. It appears that it is one person's experience that summarizes what the team has done rather than interviews with team members triangulated with quantitative data analysis.

Table 5. Books resulting from the systematic literature review, with those used when building the initial anatomy map marked in boldface.

Book [ref]	Cited	Year	Main contribution
B1 [13]	P14 P16	1975, revised 1995	Originally published in 1975, the referenced version was published for the twentieth anniversary and also includes subsequent essays on software engineering. Details experiences from the development of the IBM System/360 in the 1960s, where the author was the project lead.
B2 [57]	P1 P8	2002	Argues that craftsmanship is a better metaphor for software development than software engineering, which is described as focusing on multi-year, large-scale, low-skilled-developer projects.
B3 [47]	P3	2008	While focusing on patterns for using Scrum and Lean practices in large-scale system development, the authors also illustrate the importance of skilled developers that practice their craft, mentoring less-skilled peers.
B4 [54]	P6 P11 P17	2008	Personal experiences from the authors are combined with a set of concrete rules, exemplified in Java, to create a catalog of smells and heuristics, including remedies.
B5 [78]	P4	2008	Philosophical book, arguing that Linux and other open-source projects embody the spirit of craftsmen, as epitomized by the hymn of Hephaestus.
B6 [37]	P13 P15	2009	Originally sourced from a wiki, this book describes Software Craftsmanship as a pattern language, centered around learning themes such as "emptying the cup", "walking the long road", "accurate self-assessment", "perpetual learning" and "construct your curriculum".
B7 [77]	P11	2009	Contains 15 interviews the author conducted in 2008 with leading developers from the 1960s until today. Of the 11 interviewees asked, eight would identify software development as a "craft" Other opinions voiced were: "art", "mathematics", "science", "engineering" or "a style of writing".
B8 [55]	P6	2011	Using the author's experience as an example, describes rules and principles for professionalism in committing to a task, developing, testing, and dealing with teams and people under delivery pressure. Advocates for practicing and mentoring as tools to reach higher productivity.
B9 [52]	P6	2014	Wide treatment of Software Craftsmanship, ranging from personal experiences, professional attitude and technical practices to how to interview for recruitment and foster a culture of learning.
B10 [35]	P10	2014	Describes best practices and lessons learned while teaching the four rules of simple design[8] via code kata exercises for various groups of people over the course of five years.
B11 [88]	P6	2015	Blends the two fields of Agile software development and Human Performance Technology, a field closely related to human resources and learning professionals, described in 1978 by Gilbert[33]

Table 5 contains the books found in the SLR, with the books used by the first author to build the initial anatomy map marked in boldface. Many books (e.g., B1, B4, B8, and B9) describe personal experiences from skilled software development professionals. Others, such as books B3 and B11, detail process patterns for large-scale organizations, whereas book B7 contains transcribed semi-structured interviews with 15 senior developers, focusing on their personal development experiences and opinions. Books B2 and B5 are more philosophically inclined, and book B10 describes experiences from teaching XP and pair programming using deliberate practice.

To the best of our knowledge, this paper is the first attempt to empirically derive the anatomy of software craftsmanship based on a more encompassing view of the seminal books, supplemented by academic literature in the area, and buttressed by insights from an in-situ longitudinal industry case study.

5 THE ANATOMY OF SOFTWARE CRAFTSMANSHIP (RQ2 AND RQ3)

Our anatomy of software craftsmanship is synthesized from the case study and the SLR results. Figure 3 depicts four themes with associated principles and practices as interconnected nodes.

The ***A*** *Value-focused architecture* theme has three principles (**A1** to **A3**) with ten associated practices (**A1.1** to **A3.4**). The ***D*** *Iterative design, development, and verification* theme has three principles (**D1** to **D3**) with ten associated practices (**D1.1** to **D3.2**). The ***C*** *Shared professional culture* theme has six principles (**C1** to **C6**) with 18 associated practices (**C1.1** to **C6.3**). The ***F*** *Feedback* theme has five principles (**F1** to **F5**), with nine associated practices (**F1.1** to **F5.2**).

Some practices are connected to more than one principle, indicated in the figure via interconnected edges. Some practices are hierarchical. For instance, the practice ***F1.1*** *Requirements* contains the sub-practices ***F1.1.1*** *Accessible* and ***F1.1.2*** *Collaborative*, indicating that the requirements gathering and clarification process was performed in collaboration between the requirements engineer ("On-site customer") and the development team.

The principles are presented together with the supporting empirical findings found in the literature and the case study.

5.1 A Value-focused architecture

The software craftsmanship manifesto states as a principle: "Not only responding to change, but also steadily adding value," and a well-crafted system should have a software architecture that enables this goal.

The three principles and ten practices related to value-focused architecture are listed with references in Table 6. To enable the value-focused architecture, software architects have to participate in guiding the team into a modular and layered architecture, where changes do not ripple across subsystems, and code is kept clean and as simple as possible through refactoring. The first rule of refactoring[29] is that there must be sufficient test coverage before it occurs, so the architecture should also enable the development of a comprehensive, layered test base.

A1 Participating Software Architects

- **Literature:**
  Brooks, in B1 mentions the *chief programmer* as a role which today could be called lead software architect, and discusses the benefits of *conceptual integrity*, by using a "small architecture team." Books B2, B6, and B8 also discuss the importance of architects that participate in the end-to-end solution, for instance, by specifying and giving examples of integration tests. Looking outside the SLR scope, Hunt and Thomas [38] calls the role "technical head," tending to the big picture, and Martin [56] states that software architects need to participate in the development to spot problems and guide directions.

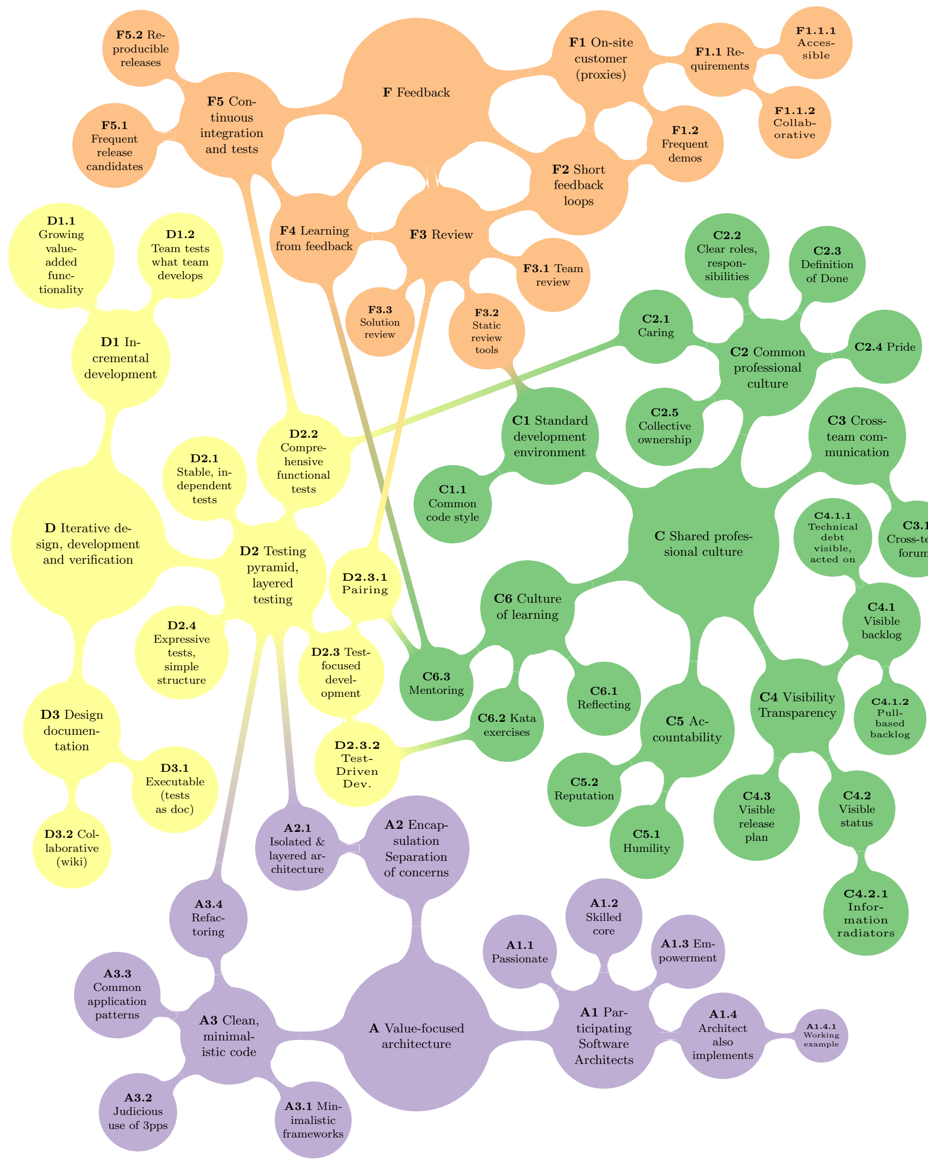

Fig. 3. The anatomy of Software Craftsmanship.

Books B1, B3 and paper P3 refer to team empowerment in the context of cross-functional teams [58], while book B9 states that an empowered a team of craftsmen can be the difference between project success or failure. Book B2 states that users should be empowered to interact with developers, who know how to use this to deliver robust applications. Paper P7 mentions the *constant attention to architectural issues* and *lead developers that participate* in the product from early prototypes to delivery. Paper P5 states that their product used an *initial domain model* and *an early definition of basic architectural mechanisms.*

Table 6. References to ***A*** *Value-focused architecture*

Id	**Name**	**Books**	**Literature**	**Qualitative**
A1	Participating Software Architects	B1, B2, B6		SwArch1, Dev1, Dev2, Dev3, Test1
A1.1	Passionate	B2, B3, B6, B7, B8, B9	P1, P8	SwArch1, Dev3, Test2
A1.2	Skilled core	B1, B2, B3, B6, B7	P1, P8, P11, P15, P17	SwArch1, Dev1, Dev2, Dev3, Test2
A1.3	Empowerment	B1, B2, B3, B9	P3	Dev3, Test1, Test2
A1.4	An architect also implements	B3, B9	P7	SwArch1, Dev3, Test1, Test2
A1.4.1	Working Example		P5, P7	Dev2, Dev3, Test2
A2	Encapsulation & separation of concerns	B1, B2, B4, B7, B8 B10	P11, P13	SwArch1
A2.1	Isolated and layered architecture	B2, B4, B6, B7, B10	P3, P11, P18	SwArch1, Test1
A3	Clean, minimalistic code	B1, B2, B3, B4, B7, B9, B10, B11	P5, P11, P15, P17	SwArch1, Dev3, Test2
A3.1	Minimalistic frameworks	B2, B4, B7	P4, P8, P11	SwArch1, Dev1, Dev2
A3.2	Judicious use of third-party-products	B2		SwArch1
A3.3	Common application patterns	B3, B7, B10		SwArch1, Dev1, Dev2, Dev3, Test2
A3.4	Refactoring	B1, B3, B4, B6, B7, B8, B9, B10, B11	P3	SwArch1, Dev1, Dev2, Dev3, Test1, Test2

The importance of skills and passion for the craft is discussed in seven books and eight papers, as depicted in Table 6, e.g., paper P2 elaborates on the role of a passionate leader in increasing engagement.

- **Empirical findings:**
  The studied system had the same chief software architect, who implemented a lot of code, including a minimal container framework, based on partial support of EJB 3 standards. "I tried not to interfere too much with the teams. Instead, I tried to ensure that the platform they were building on was stable and good enough, so whatever they did, they will most likely get it right. Because that reduces the load on me and my team."(SwArch1)
  As the product grew beyond two teams, one senior developer from each team was designated "team architect" (TA), with the intent to spread the knowledge from the chief software architect. This is further discussed in item **C3**, and similar to the one reported in [12].
  Teams were empowered to come up with their own solutions and to improve on existing solutions. The TA group also had some votes in the resource planning, for instance, regarding "onboarding" procedures for the outsourced teams, as mentioned in item **C2**.
  Several interviewees mention the passion and the pride they took in the product, e.g. "We cared a lot for our product. Some people ended up in different areas…Some features were like one's nursing child."(Test2)
  Team architects were expected to both participate in the team's daily work and mentor them into a coherent way of working: "[Our team was formed by] mixed newly graduates and senior developers. And our TA, I guess he preached a lot. He has gotten me into Domain-Driven Design. During my education years, I was using strings everywhere. So, he really opened my eyes to the benefits of DDD. And now, I try to spread the word [to my new team]."(Dev3)
  There are also contradictory views that the product was lacking a communicated architectural vision: "My dream architect should know the code, know how we want it to work, and also say 'Now when you are into this part, I want you to think about this also, improving, preparing for future… ' And also being able to delegate this."(Test1)

- **Analysis:**
  Striking the correct balance between participating and empowering is not trivial. While Bass et al. [4] do include "Implementing the product" and "Testing the product" as two of the ten technical duties of a software architect, they also list seven non-technical duties, nine non-technical skills, and ten knowledge areas that should be mastered.
  In the studied case, the developers showed lots of passion for the product and worked together towards the same goal. However, there were still expressions that there was a lack of a communicated vision and a desire for tasks and responsibilities to be delegated more.

A2 Encapsulation, Separation of concerns

- **Literature:**
  Encapsulation is the materialization of one of the most traditional Software Engineering: "the separation of concerns" [25]. While developing a complex system, there is a need to develop and evolve different parts of the system independently [4]. The layered architectural pattern is the most widely spread practice for architectural subdivision [4, 14]. The pattern segments the software systems in a way that enables modules to evolve and be developed separately so that each module has only one main reason to change.
  Five books in the SLR findings (B2, B4, B6, B7, B8, and B10) advocate proper encapsulation, loose coupling, and isolation of changes. Book B2 explicitly mentions that designing for testability is important because it discourages coupling and encourages cohesive module design. Outside the SLR findings, Richards et al. state that layered architectures increase the efficiency of testing [70]. Papers P3 and P11 state *simplicity* as a key trait of craftsmanship.
- **Empirical findings:**
  One of the first architectural decisions was to rely on an EJB 3-alike application framework, developed internally, to solve product requirements regarding installation, upgrades, and configuration. The framework is further discussed in item **A3**.
  The architecture enforced business logic to be split into interfaces and implementations and used dependency injection, using naming patterns to reduce the need for boiler-plate configuration. Inter-process communication initially used serialized Java objects, though this was later replaced with an XML-based interface, supported by a schema definition language. This change made it easier to enforce backward compatibility across different protocol versions by defining a published protocol that was shared with external parties.
  Figure 4 (a) depicts the initial layered architecture using UML stereotypes packages as layers and stereotyped *allowed-to-use* UML dependencies, as suggested in [17]. The application server is represented as a bottom layer in this figure, although it also supports all layers with cross-cutting concerns, such as transactions, security, and logging. The Data Access Objects (DAO) encapsulate the access logic to the database, and upper layers add business logic and protocol support.
  When faced with the problem of supporting clients using earlier protocol versions, the suggested solution was to add another layer in the architecture, as depicted in Figure 4 (b). The old "Operations" layer was split in two, where the new "Operations Manager" layer contained code common to the different versions of each operation, and the protocol version layer converted between the specific protocol versions and the operations layer.
  The lead system architect had strong opinions about the architecture: "If you look at each service, it has a normal, layered architecture, because everything else is wrong."(SwArch1) He also discussed the architecture's tree-based structure: "The dependencies between the different services should look like a tree because it's easier."(SwArch1)

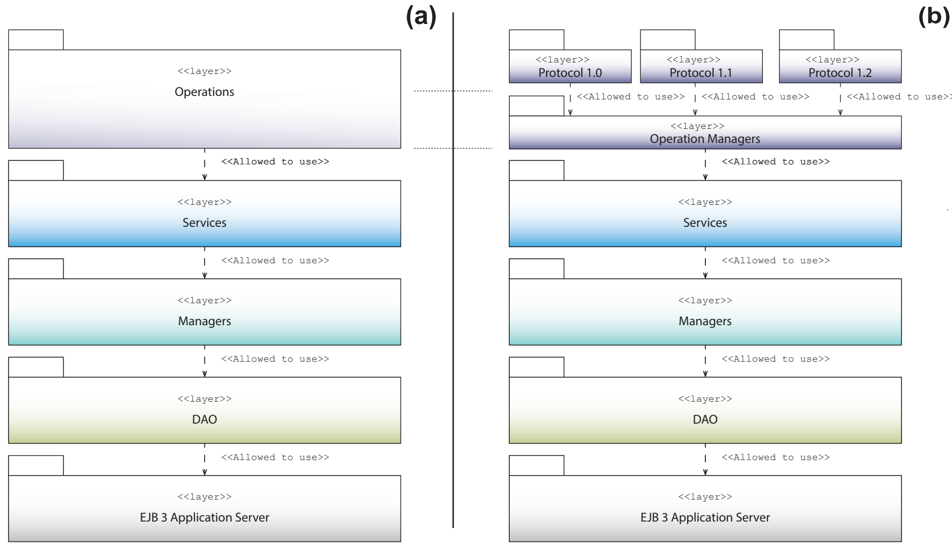

Fig. 4. Layered view of the Initial architecture (a) and Layered view of the Architecture after separating protocols from business logic (b).

Particular care was taken to separate the architectural framework from the business logic: "The bottom layer is, of course, just an interface. You don't rely on implementation because implementations can change. Then you build data access on top of that, then on top of that you build managers and compound features, and so on."(SwArch1)
Architecture should simplify the creation of business value. This includes "making it easy to make the right decisions" such as container-managed transactions and no explicit threading in business logic. It also should simplify wanted non-functional aspects, such as simple unit and integration testing, a defined data model management policy, absolute transaction security, and scaling. This was mentioned as beneficial by three developers: "There was a good framework at the product level, so you avoid doing things which are wrong."(Dev2) "[Application developers] should not need to know everything that is behind the scenes. If they need to see it, then something is wrong. Then we haven't described a certain interface good enough."(SwArch1)
The desire to simplify testing was also a driving factor: "…[listeners are used as] reversed dependency injection, to inject behavior that is needed for a particular customer…instead of trying to mush everything into the same thing. Because that will take a longer time to build, longer time to test. It will be a lot more complex to understand, and it won't be readable."(SwArch1)
Layered architecture also supported business flexibility, allowing the system to be customized for different installations while keeping a stable core. All deployments used the same core engine with customer-specific adaptations added as optional packages.

- **Analysis:**
  Following software craftsmanship principles means focusing on simplicity and testability when making architectural decisions. Similarly, the developers were supported in their evolution of the system through the hiding of unnecessary detail and having clear interfaces to features, affecting both functional and quality attributes. The architecture supported the

smooth replacement of deployed code, data models, and existing data, showing that there was a *long-term commitment* to the product.

A3 Clean, "minimalistic" code

- **Literature:**
As detailed in Table 6, eight of the studied books describe the importance of keeping the code clean and the design simple. Books B2, B4, and B7 advocate for minimalistic frameworks, and B2 also mentions that care should be taken when choosing to depend on other products. Both paper P12, and books B3, B7, B10, [56] exemplify and describe the importance of using common application patterns to communicate a design. However, in book B7, one interviewee (Brendan Eich) concedes that he never bought the Design Patterns book [31].
Nine books list refactoring as the key principle to achieve a clean codebase, indicating that clean code typically arises from successive refinements; it is not written directly. This is also stated by Hunt and Thomas [38]. Paper P3 also mentions *refactoring* as a principle of software craftsmanship. According to Fowler et al.[29], refactoring involves "improving structure without affecting existing functionality."
Papers P5, P11, P15, and P17 mention *clean code* principles, using *exploratory programming* and *reflections* to make the code cleaner.
Papers P4, P8, P11, P17, and P18 discuss how *tools are important* to a craftsman and how to *fight against homemade complexity*, using *clean abstractions*. Of particular importance is the ability to *choose the tool* based on the task at hand.
Paper P12 mentions the importance of *understanding the styles, idioms, and patterns* to be effective in a language and how the Lisp and APL communities have championed the use of *kata-like exercises* to spread common idioms for developers to be productive.
- **Empirical findings:**
Both items **A1** and **A2** mention the in-house developed architectural framework. In early 2011, the framework consisted of 299 Java files totaling 19 kLOC production code, which grew linearly (LOC p-value $< 2 * 10^{-16}$, $R^2 = 0.968$) to 72 kLOC Java production code in 1027 files in late 2016. This is clearly fewer lines of code than, for example, the JBoss (also known as Wildfly[9]) application server, which in its 7.0 release (July 2011) comprised 2886 Java files, totaling 205 kLOC, and the 10.1 release (Aug 2016) comprised 7272 Java files, totaling 433 kLOC.
The importance of the minimal framework was stated by the chief software architect: "…all these application servers, they have to support 100% of the standard. The difference with us is that we support the 5% that we need…System handling, such as installing, upgrading, configuration, and so on is usually not covered in the normal application servers."(SwArch1)
Another driving force of the framework was the ease of development: "[The foundation] is built so that it is easy to develop and debug, also locally, on your local laptop. You have the basic services, cross-functional things with interceptors, and so on. The application developer should be able to focus on the value for the customer."(SwArch1).
In the project, all interviewees mention refactoring as a used practice, though two say that it has to be "hidden" in the normal work rather than being a planned activity. One interviewee stated that refactorings larger than a week have to be planned, but smaller ones take place "in the regular feature work."

[9]https://github.com/wildfly/wildfly

Several interviewees also mentioned the desire to refactor more, to clean up more, but states that the balance tends to tilt towards finishing the current feature.
The project required developers to use strict commit messages, including the reason for the change. Possible reasons for a change included feature development, spontaneous or official (documented) bug fixes, spontaneous refactorings, or build-related changes (e.g., preparing for releases or version changes). Table 7 shows the percentages of commits of the different sorts on the master branch, not including back-ported commits to maintenance branches. The table shows that the refactoring percentage of commits varied between 27% and 7% each quarter, with both mean and median around 16%. The number of fault correction commits was lower, between 22.3% and 6.3%, with a mean of 12.6% and a median of 12.4%.

Table 7. Summary statistics of the proportion and type of main branch commits per quarter

Metric	$\bar{x}$	σ	$Q_{25\%}$	$Q_{50\%}$	$Q_{75\%}$	Min	Max
Commits per quarter	3362	1189	2699	2994	3767	1090 (Q4-2016)	6361 (Q4-2015)
Feature development	52.8%	10.6%	46.1%	54.3%	59.2%	28.8% (Q1-2011)	74.5% (Q4-2015)
Refactorings	16.8%	4.5%	14.7%	16.6%	18.2%	7.7% (Q2-2014)	27.6% (Q1-2011)
Fault corrections	12.6%	3.3%	10.9%	12.4%	13.9%	6.3% (Q4-2015)	22.3% (Q4-2012)
Build related	16.8%	6.5%	12.9%	13.6%	19.0%	8.8% (Q4-2015)	30.6% (Q4-2016)
Unclassified	0.2%	0.1%	0.2%	0.2%	0.3%	0.1% (Q1-2013)	0.5% (Q4-2013)

There were concerted efforts to clean up the code in the project and keep a consistent style throughout the codebase. As mentioned by one of the respondents, the developers should "…strive to leave the code a little cleaner than you found it."(Dev3)
In the project, several interviewees mention the help they got from the well-defined application patterns used in the product, including the security patterns (encryption, key management, and fingerprinting). "Identify the patterns. Actually, you have thousands of classes and code, but you can summarize them into one or two use cases. You need to have examples.…"(Dev1)

- **Analysis:**
  The results regarding refactoring, see Table 7, confirm that the organization was consistent in refactoring and in keeping the constant improvement culture. Both refactorings and spontaneous bugfix percentages were higher at the beginning of the project when the codebase was smaller and more volatile. However, the inter-quartile range indicates that during 12 of the studied 24 quarters, the ratio of spontaneous refactoring commits varied between one in seven ($\approx$ 14%) and two in eleven ($\approx$ 18%).
  Others have studied the effects and efficiency of refactoring operations embedded in feature development (e.g. [44, 92]).

Summary:
The architecture of a system developed with craftsmanship in mind should strive to *maximize value-creation* over a *long-term commitment* to the product. The way to achieve this is to develop and frequently validate a *comprehensive regression test base*, enabling developers to *refactor* the codebase into a *clean and simple representation*. It is as important to *care* for the test base as for the production code.

5.2 D Iterative design, development, and verification

The first principle in the software craftsmanship manifesto states, "Not only working software but also well-crafted software." The practices outlined in Table 8 are centered on verification and iterative refinement of the software and its requirements. There are also dependencies to an architecture focused on testability and clean code; as stated succinctly by Martin[55] in book B8: "The fundamental assumption underlying all software projects is that software is easy to change. If you violate this assumption by creating inflexible structures, then you undercut the economic model that the entire industry is built on."

Table 8. References to ***D*** *Iterative design, development, and verification*

Id	Name	Books	Literature	Qualitative
D1	Incremental development	B1, B2, B3, B4, B5 B6, B7, B8, B9, B10	P1, P11, P12, P17	SwArch1, Dev1, Test2
D1.1	Growing value-added functionality	B1, B2, B3, B4, B5, B7, B9, B10	P1, P3, P5, P7	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
D1.2	Team tests what team develops	B3, B7, B8, B9, B11	P3, P17	Dev1, Dev2, Dev3, Test2
D2	Testing pyramid, layered testing	B1, B2, B3, B4, B6, B7, B8, B11	P1, P3, P5, P11, P17	SwArch1, Dev2, Test1, Test2
D2.1	Stable, independent tests	B8, B9, B10		SwArch1, Dev1, Test2
D2.2	Comprehensive functional tests	B1, B2, B3, B4, B7, B8	P3, P11, P17	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
D2.3	Test-focused development	B2, B7, B9		SwArch1, Dev1, Dev2, Dev3, Test1, Test2
D2.3.1	Pairing	B3, B6, B7, B8, B9, B11	P5, P8, P10, P12, P13, P15	Dev3
D2.3.2	Test-Driven Development	B3, B4, B6, B7, B8, B9 B10, B11	P1, P3, P11, P13, P15	Dev1, Dev2
D2.4	Expressive tests, simple structure	B2, B4, B7, B8, B9, B10		SwArch1, Dev1, Dev3, Test2
D3	Design documentation	B1, B7		
D3.1	Executable (tests as doc)	B1, B2, B4, B7, B8	P4, P5, P9	Dev1, Test2
D3.2	Collaborative (wiki)		P2, P3, P4, P8, P9, P15	Dev1, Dev2, Test1, Test2

D1 Incremental development

- **Literature:**
  Ten of the studied books relate to an incremental development in some form, and the majority of them refer to "growing" software rather than "building," "designing," or "architecting" software, see Table 8. This implies that software construction is an act of successive refinement, where the software is constantly tended to and updated as the requirements or environment changes.
  Papers P1, P7, P11, and P17 discuss the *iterative development* and the *moving between designing, making, evaluating, and reflecting phases* of software development.
  Papers P1, P3, P5, P7, and P17 mention *prototyping* and how *testing is done in parallel with development.*
  Books B3, B7, B8, B9, and B11 state that teams should be cross-functional and autonomously analyze, implement, and verify functional requirements. Book B8 states: "QA should find nothing," implying that QA is a separate team, focusing on verifying other requirements than pure functions, for example, usability, stability, security, and other quality requirements of the produced system. Paper P3 also mentions the introduction of *cross-functional teams*, as one part of transforming a large organization into Lean Software Development.
- **Empirical findings:**
  Testing of functions and requirements took place in the same team, and in the same sprint, as where the development of the production code took place. Because developers using the original functional test tool could not keep up with the development pace, a couple of developers wrote a new Java-based test case runner, where functional test cases was specified in a custom XML-based language. This allowed development of test cases to proceed alongside development of production code.

Table 9 shows the linear evolution of the codebase over time for the major types of source code in the product. All studied types grow linear over time, with all p-values less than 10^{-13} and adjusted R^2 between 0.91 and 0.98. For the Java- and XML-based code, the column **Initial size** reflects the state at the start of data collection in January 2011, while the Scala-based code was first developed in Q3 2012. The column **Growth per quarter** is the calculated linear regression coefficient, and **End size** is the size at the end of the studied period, in December 2016.

Table 9. Summary code statistic for the five major code types, showing linear growth over the quarters

Code type	Lang.	Initial size [kLOC]	Growth per quarter [kLOC/qtr]	p-value	Adjusted R^2	End size [kLOC]
Production	Java	150	26.1	1×10^{-13}	0.91	753
Unit tests	Java	64	24.7	3×10^{-14}	0.92	620
Integration tests	XML	83	67.2	3×10^{-14}	0.92	1560
Web GUI prod.	Scala	9	3.3	1×10^{-13}	0.97	65
Web GUI tests	Scala	2	8.4	4×10^{-15}	0.98	129

By calculating Pearson's correlation coefficient (r_{xy}) between different types of code, we confirm that the volume of the different types of code varies together. Production code are related to unit tests by a correlation coefficient of $r_{prod,unit} = 0.998$ (p-value $< 2 \times 10^{-16}$), and to integration tests by $r_{prod,int} = 0.996$ (p-value $< 2 \times 10^{-16}$). The web GUI production code are related to the web GUI tests by $r_{webprod,webtest} = 0.975$ (p-value $< 6 \times 10^{-12}$).
All interviewees mention the highly iterative development process, and one developer contrasts this with regular consultancy work: "In a consultancy, they focus more on the delivery than on the craftsmanship…We used an iterative, test-driven way, to be prepared for what can happen in the future."(Dev1)
Several interviewees also mention tests being developed alongside the production code, e.g., "We used to ensure that whatever test cases had been written in the [test plan, a shared Excel document] will translate into some automated test cases."(Dev2)
"A strength was that we could test in parallel with development, based on a clear requirement base, in [the wiki-based requirement tracking system], where everyone could read it."(Test2)

- **Analysis:**
  Incremental development is part of getting reliable and actionable feedback and so is tightly tied to **F2** Short feedback loops. Because the teams owned "the whole development process," including functional testing, they took responsibility for the entire development phase, including documenting used solutions.
  The fact that all five types of code grow linearly, together, indicates that software was developed incrementally throughout the studied period. In a non-incremental scenario, we would have expected integration tests to lag behind the production code as the focus of the organization moved to test phases that followed growth of production code and unit tests. We see no such findings in our data.
  The organization took action when it discovered that the originally used functional test tool could not keep up with the development pace and created an alternative solution based on structured text files. However, the amount of function test code soon eclipsed the production code, and it continued to grow faster throughout the study.

D2 Testing pyramid, layered testing

- **Literature:**
  Eight books and five papers stated that tests should be layered into different categories, see Table 8. The importance of having a stable base of test cases, independent of each other, is mentioned in three books, B8, B9, and B10.
  Papers P3, P11, and P17 mention how *solutions can be proposed by writing tests*, for instance, using Behavior-Driven Development, and the practice of high-level integration tests is also stated in books B1, B2, B3, B4, B7, and B8.
  Focusing on the development of tests, whether using *Test-Driven Development (TDD)* or a less stringent method, is mentioned in nine books and six papers, with papers P3, P13, and P15 explicitly mentioning *TDD* as a craftsman skill to practice.
  The practice of having automated tests of different kinds with a readable, simple structure is stated in five books, with the most pointed citation mentioned in book B8: "Unit tests and acceptance tests are documents first, and tests second. Their primary purpose is to formally document the design, structure, and behavior of the system."
  The "Agile Testing Quadrants" model [21] can be used to classify tests along the lines of "supporting the team" and "criticizing the product," versus "business-facing" (verifying customer requirements) and "technology-facing" (verifying individual implementation decisions). Outside the SLR findings, the books [38] and [56] also state that designing for testability increases the likelihood of tests being developed.
  Paper P5 explains how a *successful test run triggers a new executable package and deployment* to a DevOps pipeline, followed by further non-functional testing and further validation.
- **Empirical findings:**
  In the studied case, already from the start of the product, considerable focus was placed on verification on several layers, as illustrated by the test pyramid [18]. While some developers preferred Test-Driven Development, others instead preferred to write tests after the production code, but tests were expected to be developed close to the production code, minimizing feedback time (item **F2**). As stated by the lead architect: "I call it *Test-Focused Development*, because one of the ground rules is that, if you build something, it should be easy to test. Always easy to test…If it is easy to unit test and function test, then it is better than building the small, slimmest solution. So, I always have this pyramid…You should work with tests from Day 1. If you don't do that, you're doing it the wrong way"(SwArch1)
  Another interviewee confirmed the test focus, by comparing with another product: "I think it [relates to] how we introduced ways of working in [studied case] We focused much on test coverage, and there was solid practice related to which test cases to write, how to review and present them. There was much more focus on testing, on automation and those areas."(Dev2)
  The amount of (functional) integration test code soon eclipsed the production code, while the unit tests grew at the same pace as the production code. The same pattern repeated itself when the new Scala-based web-GUI was developed in 2012, as its functional test codebase, also written in Scala, grew faster than the GUI production code.
  Figure 5a shows the numbers of non-commented source code lines for the production code (*prod (Java)*), unit tests (*unit (Java)*), integration tests (*int.test (XML)*), web GUI (*prod (Scala)*) and web GUI integration tests (*int.test (Scala)*), and Figure 5b shows the relative size of the unit tests and integration tests versus the Java production code, and the relative size of the Scala-based integration tests versus the Scala-based GUI production code.
  The figure shows that the integration tests were growing much more than the production code, while the unit tests kept approximately the same growth rate. As reported in Table 9,

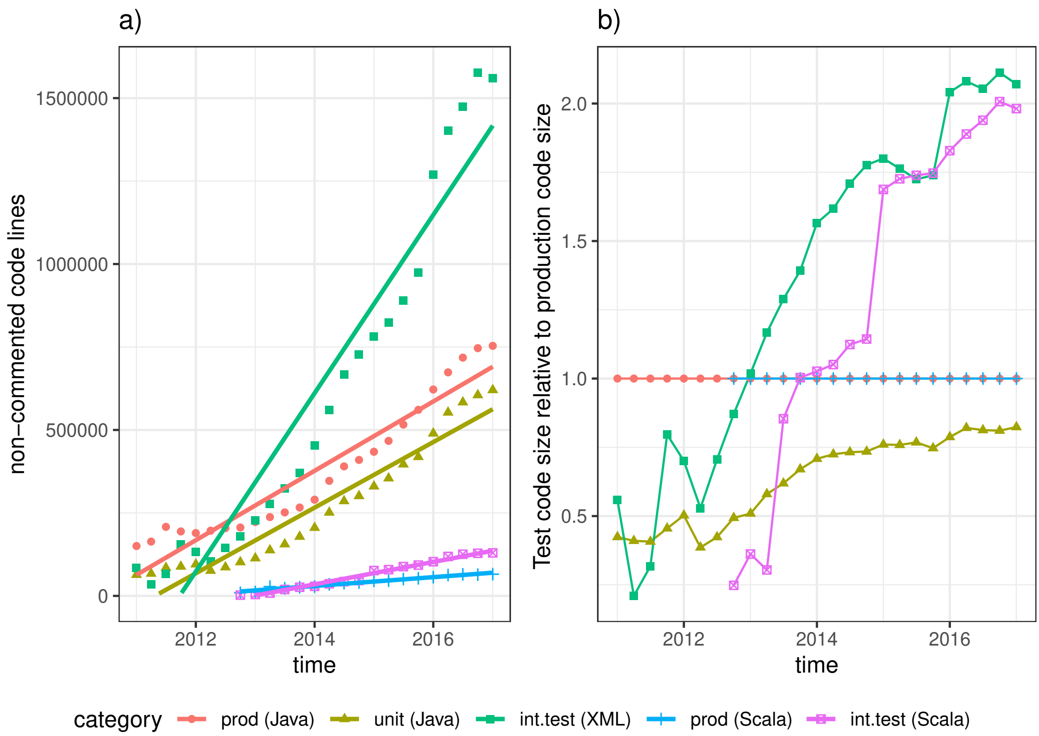

Fig. 5. Ratio of test code versus production code over time.

all five codebases grew linearly throughout the studied period. The three dips in integration test size between Q1 — Q2 2011, Q1 — Q2 2012, and Q4 2016 — Q1 2017 were due to product realignments, where old protocols and functions were removed. Both integration tests (written in XML) and GUI tests (written in Scala) grew to about twice the size of the corresponding production code, although the GUI code was much smaller. The unit test base was initially slightly less than half the size of the non-GUI production code but grew to about four-fifths ($\approx$ 80%) of the production codebase.

The unit tests can be further subdivided into “pure unit tests” (no interaction with the outside world) and “fixture tests,” where the tests interact with a locally installed and prepared database. Non-functional testing used dedicated hardware, including dedicated simulators. The product placed a relatively large emphasis on unit tests that interacted with a locally installed database, using the *Transaction Rollback Teardown* pattern[59]. At the end of the studied period, 8327 integration tests, 18412 database-interacting unit tests, and 5328 “pure” unit test methods had been developed. The number of pure unit test cases were higher, as these also included parameter-driven tests generated from the code via reflection, see item **F3** about the “meta-tests.”

Each developer knew how to use and develop integration tests, though, in practice, one or two persons per team focused on writing them. “Anyone should be able to do the testing. . . One or two persons in the team, part of the team, developing [integration] test cases. He used to get assistance from other developers, in case required.”(Dev2).

Another developer mentions, "…some testers might not have the correct background or understanding, so I gave them a template, like: 'This is how I think, now you explore more into your scenarios…'"(Dev1)
The lead architects decided to include "test helpers" in the functional verification phase, which facilitated efficient integration testing. "And then add some test packages on the side, which are used in the testing. So it's not black-box, but more gray-box. You use those packages to make your test flow a little better."(SwArch1)
The *Definition of Done* for feature development (see item **C2.3**), stated that functional verification should be automated before feature delivery. How to achieve this was regularly discussed in cross-team forums (see item **C3**). "Everything should be tested, and there should be automatic test cases for everything…"(Dev3) Despite this, some manual functional test cases still existed. At the end of the studied period, there were 24 documented manual functional test cases, mostly related to data aging (importing/exporting archived database data) or security issues. These were executed based on a "risk-based judgement," typically when changes had been made in the tested area or before major releases of the product. The system testing team also focused on manual testing, such as validating instructions for administrators or integrators. This test phase was the first with a full hardware deployment, including Hardware Security Modules, application firewalls, and load balancing hardware. In contrast, functional testing in development teams utilized plain Linux virtual machines. One developer mentions that the team structured their work so they would interact all the time and used this as a form of pair programming: "We did not divide tasks [in functional areas], such as GUI, persistence and so on. Instead, we pair-programmed a lot. We were encountering each other's code all the time, communicating verbally: 'Hey, this method you did — can I change it, make it better?'"(Dev3)

- **Analysis:**
Specifying requirements as test cases will lead to the volume of test code eventually outgrowing the production code, as is visible in Table 9 and Figure 5. Therefore, it is important that these test cases (requirements documents) are easily readable, frequently maintained and executed to ensure that they still reflect the state of the product. Bjarnason et al. [8] describe five different variants of using test cases as requirements, based on a multi-case study made at three companies of various sizes. In particular, while the largest company had failed to completely specify end-to-end behavior, including user interactions, as test cases, they reported success in using the process for API development.
Having this layered testing architecture as a regression test base enables safe refactoring and transformation into clean code (item **A3**). Thus, the test base enables clean production code, and the tests are required to be clean in order to be readable and maintainable. Overall, this enables an evolutionary growth of the software, without "big-bang" integration phases. However, cleaning and refactoring the tests themselves are harder to achieve. When changing test cases, care must be taken that the changed tests cover the original requirements. How to achieve this remains an unanswered question.
There will always be some tests that are not possible or economically viable to automate. In the studied product, the developers identified 24 test cases out of a functional regression test base of 8327 test cases (0.29%) as belonging to this category.
The different layers of tests are important to enable the feedback loops necessary to guide incremental design and development. Each layer has different trade-offs related to reflecting the true production environment behaviour versus being fast and efficient to develop and trouble-shoot. In the product, many unit tests interacted with a locally installed database, which has the disadvantage of adding lead time to the feedback loop. However, there is

also an advantage in that relatively large parts of the system can be tested down to the SQL level without mocking behavior.

D3 Design documentation

- **Literature:**
  Five books mention documentation in relation to craftsmanship, as *self-documenting programs*, in B1, *tests as documentation*, in B4 and B8, and B7 references to Knuth's work on *literate programming* [45]. Book B2 states that "a lesson from software engineering is that hardware and software never quite match their documentation." One solution to this proposed in both B7 and [38] is to extract documentation from the source code.
  Papers P2, P3, P4, and P15 mention *collaborative documentation* through Wikis or shared recordings. Paper P9 states that a *shared user story repository* gives immediate feedback on changes. Papers P4, P5, and P9 mention *code as communication*, exemplified by Domain-Driven Design, and acceptance tests in the form of *executable user stories*.
- **Empirical findings:**
  The studied product had no formal design documents (e.g., component descriptions) maintained by the development teams. Instead, they relied on a wiki system to document design principles and executable test cases as documentation of required behavior. The organization used deliberate practice (see **C5**) as a tool to teach development principles.
  As part of defining the external API, a tool was developed based on the *Javadoc*[10] tool, converting code comments and annotations, including validation rules, into a form suitable for customers or system integrators. This documentation evolved together with the API.
  The integration test cases also frequently served as documentation of how the product behaved, putting pressure on their quality and descriptions. The test case structure, including directory and file names, became part of the documentation, as it became harder to know where to look as the test base grew. As discussed in item **F5**, the automated test cases were continuously executed, and their results verified, meaning that the current tests reflected the actual state of the product.
  Several interviewees mentioned that they were using tests as documentation: "The test was the documentation…even if we had followed [the requirement tracking tool]."(Test2)
  One interviewee mentioned the lack of design documentation as a hindrance: "There are different levels of documentation. There are many complaints [from developers] that, for instance, data models are not documented, there is a lack of a leitmotif. On an overarching level, to get the big picture, there is quite good product documentation, though.."(Test1)
- **Analysis:**
  Executing design documentation towards a working system means that inconsistencies quickly surfaces, enabling quick corrections. However, as the test base grows, the internal and external structure becomes extremely important. Each test case needs to be self-sufficient, describing its needed environment and its setup. Business-facing tests should be specified in an appropriate high-level language, such as a Domain-Specific Language, to be accessible to people not directly involved in development.
  Collaboratively edited wiki pages documented the core design principles, with automatically executed test cases documenting the detailed product features. Documents targeted for customers or support personnel were kept at a high functional level. Detailed protocol documentation was generated directly from the source code, so it would automatically match the delivered product.

[10]https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/doclet/overview.html

We see some evidence that there was a perceived lack of certain aspects of documentation, though the overall product level seems acceptable. This could indicate that having a more structured approach to design documentation than a wiki system could have long-term benefits. At the same time, we see that developers were using the test base as documentation, meaning that as long as the tests are readable and at an appropriately high level, the system's code and behavior would be understandable.

Summary: When focusing on incrementally growing software, it is essential to focus on, and build, a comprehensive regression test base to validate that what was built still adheres to prior requirements. The regression tests will serve both as a safety net and as the actual specification of the behavior of the system under construction. As such, they should be readable by both programmers and requirement owners. To meet this goal and to ensure quick feedback, tests shall be structured in different layers. Higher-level tests shall use a language closer to the business domain than the ordinary programming language to support its usage as system documentation.

Note that not only the tests but also their organization and structure act as documentation. This is because the volume of tests will eventually eclipse the production code, and all developers should realize that it is as important to work with and care for the tests as with the production code.

5.3 C Shared professional culture

The software craftsmanship manifesto states: "Not only individuals and interactions, but also a community of professionals," as well as "Not only customer collaboration, but also productive partnerships," which implies a long-term commitment to what is produced.

The focus on the community of professionals also implies a shared, common culture. As illustrated in Table 10, we have found evidence that a shared culture of *learning*, *caring*, *accountability* and *transparency* is beneficial and aligns with the craftsmanship approach.

C1 Standard development environment

- **Literature:**
  Books B1, B2, B6, B7, and B9 mention the benefits of standardizing on a toolchain. In particular, book B2 notes that the partnership approach highlights the importance of focusing on long-lived development tools.
  Four books (B4, B7, B9, and B11) explicitly mention how shared coding standards help communication and readability. Brad Fitzpatrick, in B7, mentions how Google keeps strict guidelines for programming styles, including code layout, formatting, naming, and which patterns and conventions to use[11].
  Several papers also promote common development standards as beneficial for software craftsmanship in terms of *structured exercises* to learn the correct shortcuts for the particular tool in use (P12), improve *source code quality* (P5), the usefulness of a *wiki page* containing both *coding style* guidelines as well as instructions for *how to set up the environment* (P2), capturing *IDE configuration* in a repository (P9), creating a sense of *commitment to a particular tool* (P4) and *obtaining necessary knowledge* how to best use or not use the latest *technologies, tools, processes, and practices* (P8).
- **Empirical findings:**
  At the start of product development, the lead architect chose a shared development style and code rules. The unified style helped both understanding the code and aided in merging and back-porting fixes to older branches.
  Although standardized, the used toolchain varied over the years. Initially, developers used Eclipse on Windows laptops, later also IntelliJ and Linux laptops, and eventually, Windows

[11] https://github.com/google/styleguide

Table 10. References to ***C*** *Shared professional culture*

Id	Name	Books	Literature	Qualitative
C1	Standard development environment	B1, B2, B6, B7, B9	P2, P4, P8, P9, P12	SwArch1, Test1
C1.1	Common code style	B4, B7, B9, B11	P2, P3, P4, P5	SwArch1, Dev1, Dev3
C2	Common professional culture	B2, B7, B8, B9	P3, P7	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
C2.1	Caring	B3, B4, B8, B9	P1, P3, P7	Dev1, Dev2, Dev3, Test1, Test2
C2.2	Clear roles, responsibilities	B3, B8	P3	SwArch1, Dev1, Dev2, Dev3, Test2
C2.3	Definition of Done	B3, B7, B8, B9, B11	P3	Dev1, Dev3, Test2
C2.4	Pride	B5, B6, B7, B8	P4, P17	Dev3, Test2
C2.5	Collective ownership	B7, B8		SwArch1, Dev3
C3	Cross-team communication	B3, B7, B9	P1	Dev1, Dev2, Dev3, Test2
C3.1	Cross-team forums	B3, B9	P1, P2, P3, P4	Dev1, Dev2, Dev3, Test2
C4	Visibility / Transparency	B1, B3, B6, B7, B9 B11	P3, P9	
C4.1	Visible backlog	B3, B9, B11	P3	Dev2, Test1, Test2
C4.1.1	Technical debt visible, acted on	B9, B11	P5	SwArch1, Dev1, Dev2, Dev3, Test1
C4.1.2	Pull-based backlog	B3	P3, P5	
C4.2	Visible status	B3, B8, B9	P3, P9	Test1, Test2
C4.2.1	Information radiators	B3	P3, P9	
C4.3	Visible release plan	B9	P3	Test1, Test2
C5	Accountability	B2, B3, B7, B8, B9	P3, P8	Dev3, Test1
C5.1	Humility	B6, B8		Test1
C5.2	Reputation	B2, B6, B7, B9	P2, P8	
C6	Culture of learning	B1, B2, B3, B6, B7, B8, B9, B11	P3, P11, P12, P15	SwArch1, Dev1, Dev3, Test1, Test2
C6.1	Reflecting	B2, B3, B6, B9, B11	P1, P3, P5, P11, P15, P17	
C6.2	Kata exercises	B6, B8, B9	P3, P10, P12, P13 P15	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
C6.3	Mentoring	B1, B2, B3, B5, B6 B7, B8, B9	P1, P3, P8, P15	

was dropped as a development platform. Costs and competence were cited as the reason for changing both IDE and OS. When the vendor released a usable IntelliJ version free of charge, the perceived benefits (relative to the already free Eclipse) outweighed the cost of change. Similarly, when the company introduced Linux laptops as a supported development environment, the organization quickly adopted the new development platform, as it allowed developers to develop software in an environment close to the target environment, which always was Linux. When introducing the new IDE, it was configured to format code in the original Eclipse formatting style.

The lead architect switched build tool from Apache Ant to the more expressive Gradle tool in mid-2012. The decision was driven by the new tool's stricter dependency management, stricter build scripts, increased performance and the ability to more easily develop plugins. The new tool was used to automate more release tasks, and to build a domain-specific language (DSL) for deploying test machines in different configurations, resulting in more varied automated integration testing. As stated by the lead software architect: "Large-scale software development requires both structure and flexibility, but these must never cancel each other out. I think Gradle performs a better balancing act than, for example, Maven and Ant, which are at the opposite ends of that spectrum."(SwArch1)

The Eclipse formatting rules were added to a shared repository in November 2011, as part of the first expansion to a remote site. Until then, developers used the standard Eclipse configuration. In January 2016, a similar ruleset was created for IntelliJ.

- **Analysis:**
  The standardized code style is beneficial for sharing code between the different branches as it helps version control tools merge code automatically, without distracting white-space or formatting differences. Having a shared toolkit also helps people understand and be more efficient in helping each other.
  As evidenced by the empirical findings, standardized tools do not imply a static toolkit. Instead, a learning organization should always be on the lookout for new and better tools that do the task at hand more effectively and efficiently. However, the cost of changing tools will include teaching the organization ways of working with the new tool.
  Some tools are more challenging to switch than others. While the swap of the build tool Ant for Gradle involved few persons and was made abruptly, switching development IDE from Eclipse to IntelliJ took much longer and included trying to configure the new supported IDE so it could peacefully coexist with the older supported tool.
  Static code analysis and having the build process fail in case of violations helped unify the code style, as described in item **F3**.

C2 Common professional culture

- **Literature:**
  While Boehm in P14 expresses a view of "software crafting" as the "cowboy programmer," who "hastily patches faulty code by pulling an all-nighter," this is not the dominant view in the surveyed literature. Instead, four books (B2, B7, B8, and B9) expressly state the importance of teamwork and how important it is to create a common culture of collaboration. This view is also expressed in P3 and P7.
  Four books, B3, B4, B8, and B9, state the importance of caring for the test suite (the "code production line"). Hunt and Thomas [38] also mention the *broken window theory*, first formalized by Wilson and Kelling [43], and how it relates to the importance of keeping the test base clean and working at all times.
  Any organization larger than an individual would benefit from expressing the expected roles and responsibilities. Larman et al. in B3 recollect how one chief architect states that Scrum helped the team take responsibility for their assigned tasks. In B8, Martin expresses the view of having separate, but jointly collaborating, QA and development teams. Paper P3 reports how *Communities of Practice*, together with open spaces, support discussing problems, solutions, and new ideas regarding a specific role, practice, or topic.
  Five books (B3, B7, B8, B9, and B11) and paper P3 explicitly mention the concept of *Definition of Done (DoD)*, relating to a Scrum practice. Paper P3 refers to the DoD as partially standardized, while book B8 implies that the actual DoD would vary according to the business requirements, which analysts should write as acceptance test cases.
  To take pride in one's work is mentioned by four books (B5, B6, B7, and B8) and two papers (P9, P17), and both Martin in B8 and Hunt and Thomas [38] states how this is related to *responsibility* and *accountability* (**C5**).
  The principle of *collective code ownership* is a loaded term with multiple views present. Two experienced interviewees in book B7 lean towards individual code ownership as something that cannot be denied, while Martin, in book B8, states that it is better to break down all walls of code ownership and have the team own all code.
- **Empirical findings:**
  In the studied case, all lead developers had prior experience working with overseas teams. For this reason, they requested that teams onboarded from China (in 2011) and India (in 2013) were to visit the primary site for several months to learn the product and the

professional culture, in particular, the product development process, including team tasks, planning, and verification. When the Indian teams went back to their site, a senior developer joined them for a year to support and guide their development efforts.
The studied organization had a shared "Definition of Done" with clear and actionable checks in several areas, such as *Requirements*, *Security*, *Design*, *Test*, and *Customer Documentation and User Experience*. Three different checkpoints were in place:
– End of initial requirements gathering → start of product development
– End of product development → start of system testing
– End of system testing → feature released to the market
Each checkpoint had a template-based DoD checklist, signed off before the feature moved to the next phase. The requirement engineer (Product Owner, see item **F1**) signed off the initial checklist. The Scrum Master in the development team signed off the middle checklist, and the team lead in the System Test team signed off the final checklist.
Developers from both the primary and secondary sites indicate that they felt a similar mindset in both sites. "…work culture in [main site] and India was almost similar…But in [other product] I see lots of difference between every corner of the world."(Dev1)
The developers also appreciated the practice they received and the concrete principles they learned. "…entering into a project with solid principles, these are the layers, with full hands-on experience, was the best."(Dev1) "You have a defined way of working, with respect to how you code the application."(Dev2)
Two interviewees mentioned the pride they took to make sure that what the team produced should also work. "We had some kind of pride in the team. We don't hack together something and just leave it. Rather, when we say that we are done, then we really *are* done.."(Dev3)
The regression test suite was provided with constant attention and care. To counter instabilities, in 2015, the organization set up a separate daily meeting with a participant from each team, discussing unstable or erroneous test cases and distributing them between teams. As described in item **C4**, the teams distributed and managed the identified tasks.
The test code was seen as important as the production code, as this was the documentation of how the system should behave. "The test code was equally important as the production code, because the tests showed what the product could do, like a fact-based answer."(Test2)
Two interviewees also mentioned how all developers cared to avoid security vulnerabilities in this product, relative to other experiences: "[In this product] there was a common way of working, focus on security, risk review, code reviews…These were very good controls. But when I moved to [other product], they did not care about anything…Dev2"(.)

- **Analysis:**
The surveyed literature indicates that the "lone cowboy programmer" view of software crafting has little support by practitioners, which also is implied by the manifesto focus on "a community of professionals."
The concept of *Definition of Done (DoD)* has been studied before [79] and is well-known in a Scrum context. According to the study, the focus of the DoD should be on the *systematic requirements* that are *common* for each user story. The studied organization followed this approach, using three different DoD checklists, corresponding to the three development phases (elaboration, implementation, and system testing) before a feature was released.
It is undoubtedly the case that developing a large regression test base requires care and thoughtful design of how to prevent instabilities. For developers to trust that the tests reflect the true state of the application, the test base needs to be stable and predictable.

C3 Cross-team communication

- **Literature:**
  Two books (B3 and B9) and four articles (P1, P2, P3, and P9) mention the importance of communication across teams, for instance, using the concept of *Communities of Practice (CoP)* [86]. These communities are used to source and validate potential solutions, spread knowledge, and instill and reflect upon social and professional norms.
  Paper P2 explicitly states that the studied organization had tens of different CoP, which formed as needed and ceased to work when they were either dysfunctional or had fulfilled their purpose. The paper also states the importance for a CoP to have a *good topic*, *passionate leader*, *proper agenda*, *decision-making authority*, *open communication*, *suitable rhythm*, and *cross-site participation*, where applicable.
  Different Communities of Practice, known as "Guilds" within Spotify, their challenges and benefits, have also been studied before, e.g. [1, 85].
- **Empirical findings:**
  In order to establish a common way of working, one developer stressed the cooperation that took place between teams: "It was not unusual to work across team boundaries when working with the test cases. When we discussed and found that the structure would not hold any longer, we discussed how to set the new structure. And then two or three participants would do the actual restructuring and report the progress on our [QA group] meetings."(Test2)
  Indeed, as the number of development teams grew in the product, a need for more efficient communication surfaced, both for architecture and testing activities, causing the organization to establish both a Team Architect (TA) group and a Quality Assurance (QA) group. Each group contained one member from each team, meeting regularly, the TA group twice, and the QA group once per week.
  Four developers mentioned the value of the recurring reviews as a means of competence sharing, for instance: "We used to present how we would implement a particular requirement [in the TA group] and get feedback. A very structured approach."(Dev1)
  "Having coverage — what do we think we need to do? So, implementations were reviewed in the TA forum, and test analysis in the QA forum. Where the other teams could give their feedback. You explained what you intended to do, and they could comment: 'No, but you missed this area' — because they might have worked in that area recently, and we had never been there."(Test2)
  One interviewee mentioned that time-boxing was used to limit the amount spent in meetings: "When we grew with more teams, we had to split up in review-groups, to review each others' [analyses] in detail. Building those groups based on competence to get good competence spread. [In the meetings] we made sure that everyone had read the analysis before the meeting, to be efficient, so we just could focus on the comments [that all members provided]. Sometimes we had mail conversations in these groups as well. But the analysis was documented [on the shared wiki]."(Test2)
  In the studied product, each TA member had 20% of their time allocated for TA related improvement tasks, and a similar agreement existed for the QA group.
- **Analysis:**
  Our evidence supports the benefits of Communities of Practice (CoP), both in spreading knowledge (e.g., via review feedback) and professional norms (e.g. amount of tests needed). Participants from both the primary and the remote site participated in the weekly CoP meetings, ensuring that the communication flowed between the sites.

C4 Visibility / Transparency

- **Literature:**
  The principles of visibility and transparency are closely related to the **C5** Accountability and professionalism inherent in *productive partnerships*.
  Keeping the product backlog visible and up to date is mentioned in three books (B3, B9, and B11) and paper P3. The Lean principles of *keeping options open* and *limiting work in progress* by having a pull-based backlog are mentioned in papers P3 and P5 and book B3.
  The importance of visualizing and acting on technical debt is also mentioned in the books B9, B11, and in paper P5.
  Being open and clear about development status is discussed in three books (B3, B8, and B9) and in papers P3 and P9, where the goal of *maximum project status visibility* is stated. These two papers and book B3 also highlight how the use of *information radiators* helps in this regard.
- **Empirical findings:**
  As described in item **C3**, the studied organization formed cross-teams forums to counter the blame game often surfacing before meeting a deadline.
  One identified problem was the large test base (shown in Figure 5), which required continuous maintenance effort. As described in items **C2** and **C3**, starting in 2015, teams coordinated to discuss, distribute and solve issues in this test base. The QA group was also driving improvements in this area, acting as a discussion board and mentoring others.
  Information radiators in the team area, initially two lava lamps, later replaced with nine remote-controlled LED lamps, were used to broadcast the most important build status.
  Stressing to make deadlines often cause people to take shortcuts. One often-used shortcut was to tag failing or unstable test cases as *Ignored*. The team mitigated this behavior by using Git logs to determine who had ignored a particular test case. After an initial grace period, automated periodic reminders were sent to this author to either fix or remove the test case. The QA forum discussed and took decisions on how to proceed with such tests.
  "Sometimes you had to go in and ignore test cases. . . And later, you got an automated mail, stating, 'Please fix. . . ' By then, you most likely had forgotten about the ignored test case, so you had like a 'reproach' there."(Test2)
  Several interviewees mentioned the importance of visibility, of being honest about the status and potential obstacles, and being aware of the planned releases. "Having a dialogue, saying 'No, we are not done yet, because. . . ' and highlighting potential delays as soon as possible. I think that was a strength also, to be able to de-scope, moving to a later feature. We never skipped [particular phases, e.g., testing], but rather whole areas or scopes.."(Test2)
  One interviewee mentioned a particular strategy for dealing with project managers, who tend to prioritize delivery precision over delivery contents or quality: "A senior developer taught me to frame estimates like: 'If I am allowed to do this task, it will take me four weeks. But if we don't do it, the cost will be eight hours per week, per team, indefinitely.' If you start to present those estimates, then [the project manager] will act."(Test1)
  Many interviewees also mention that refactorings **A3.4** were important to manage the technical debt: "The best part was that technical refactorings were taken as kind of a task, whereas in [other product] it is taken as a feature, and nobody will budget for it.."(Dev1)
  "The legacy that exists that is extremely large. . . You always build a little debt. But you always need to *know* what your debt is. And work with it continuously.."(Dev3)
  "Of course, we would like to refactor more. But I still think that we get a reasonable time for it.."(Test1)

- **Analysis:**
  As stated above, visibility and transparency are closely related to the principle of *productive partnerships*, where the long-term commitment is seen as more beneficial than the deadline-driven urge to "patch together something."
  The concept of *Technical Debt* [22] was created as a metaphor to illustrate when developers choose or are forced to take shortcuts, such as ignoring test cases. It is important to keep track of such debt, and the studied organization used automated tools to remind the author to take action (i.e., consider how to proceed with the ignored test case).

C5 Accountability

- **Literature:**
  Showing accountability for what you produce is mentioned as a craftsmanship trait in books B2, B8, and B9. In B7, Joshua Bloch states that "ultimately, you are responsible for your own work." Hunt and Thomas [38] also note that a professional software developer should expect to be held accountable and honestly admit mistakes or errors in judgment, which also plays into item **C4**. Paper P3 also mentions team accountability, whereas paper P8 stresses personal responsibility and sound work habits as characteristics of successful craftsmen and -women.
  Books B6 and B8 stress the importance of humility to counter professional pride. In B6, the authors argue that apprentices should combine humility and ambition to progress in the right direction. In B8, the author stresses the importance for all professionals to show both pride and humility.
  Reputation as a basis for recruitment and professional career are elaborated in four books (B2, B6, B7, and B9) and papers P2 and P8. Paper P8 argues for adopting a value model where software leaders have key qualities, such as *a proven track record* and *a personal approach to solving problems that imparts a signature to their work*. Paper P2 refers to how participation in a Community of Practice enhances professional reputation.
- **Empirical findings:**
  As mentioned in item **F2**, the project relied on releases built strictly from version-controlled files, including the build system itself. Published code artifacts were signed by each developer using their private key, and the signature was validated towards an application-specific Certificate Authority (CA) at runtime. Components were published by individual developers, while the composite release was assembled and published by a dedicated Build Master role, rotating among senior developers, allowing developers to establish a reputation and enforcing traceability towards accountability.
  One developer mentions that team accountability and pride were used to counter the pressure from other stakeholders to "just get it done." Another developer stresses the architects' accountability and responsibility to communicate a vision of the direction.
- **Analysis:**
  Accountability and responsibility are loaded terms but have long been standard practice in successful open-source projects, such as the Linux kernel, where no code is merged or released without proper sign-off by a responsible release master. These are also highly linked to item **C4** Visibility / Transparency, implying that participants should take responsibility for their creations, highlight issues and learn from mistakes, rather than place the blame elsewhere, which is typically the case in dysfunctional organizations [87].

C6 Culture of learning

- **Literature:**
Eight books from the SLR findings state the benefits and necessity of a culture of learning and continuous improvement, which clearly is a major part of software development. Five of these, B2, B3, B6, B9, and B11, also state the importance of reflecting on improving efficiency and becoming a reflective practitioner.
Papers P3 and P5 stress the notion of *learning from feedback*, such as first-hand evidence or team experiments. Paper P11 calls for *ongoing move-testing-experiments*, where bugs are seen as talk-backs from the material that drives the development process forward. Paper P2 focuses on *knowledge sharing and learning* as a part of Communities of Practice. Paper P15 fosters *self-directed learning skills*. Papers P1, P3, P5, P11, P15, and P17 all mention the importance of *reflecting and improving processes*.
Three books (B6, B8, and B9) and five papers (P3, P10, P12, P13, and P15) describe using reflective practice via *kata exercises*, sometimes practiced in a *coding dojo*. Paper P12 relates the kata concept to "experience levels," and paper P10 draws conclusions from data gathered during a *global day of kata exercises*.
Eight books describe mentoring, with B5 vividly describing how the medieval master craftsman Antonio Stradivari failed to pass on his violin-making secrets to his sons, either because he could not mentor them or because he was not aware of them. Papers P1, P3, P8, and P15 mention the importance of *coaching and mentoring* as craftsmanship principles.
- **Empirical findings:**
Learning culture was embodied in the project via a set of exercises called code katas, which explained and showed how to use the product development framework to develop functionality with the tests in focus using Test-Driven Development (TDD). The katas were first developed in 2013, preparing for expansion to the India site, and were updated as the product framework evolved. Eventually, ten katas were developed, building a simple Java application from scratch to a fully-fledged GUI, using Scala and the GUI framework used in the product. The katas built on each other and, depending on the team's experience, took between one and two hours each to complete.
The first couple of teams performed the exercises in a group setting. While this was time-consuming, it also helped the team members to learn about each others' strengths and weaknesses and support each other. Throughout the studied period, newly onboarded developers used the katas to learn how to develop in the product framework. Unlike the initial sessions, these exercises were done individually or in pairs, shifting the learning experience more onto the individual.
During the initial years, sprint demos for the entire development organization were used to spread knowledge and show newly developed features. As the number of people grew, this became too cumbersome, and the cross-team forums were used instead to spread knowledge.
"I think those mini-demos we had [in the beginning], for the whole organization, was a way to spread knowledge…Really important also that even though we worked in teams, the decisions we made were shared among the teams [in cross-team forums]."(Test2)
All interviewees mention the katas and agree that they were a vital teaching device.
"It was a straightforward, focused approach. During the kata sessions, I realized that [in my team], we have different people with different backgrounds…I could see what mistake that they were doing and I could coach them…."(Dev1)
"One way of practicing is doing structured practice…Just to learn the IDE shortcuts."(Dev1)
"…always try to stay ahead of everyone else…It's better to fail, and learn something, than not try at all."(SwArch1)

Two interviewees mentioned retrospectives as a way to reflect on their progress: "We used to do retrospectives after each sprint, where we realized: 'OK, we had this problem in this delivery — how can we avoid it the next time?', and we used to collect this in an Excel file to aid the next task."(Dev2)

- **Analysis:**
  As Brooks stated in B1, software developers are expected to learn new techniques and tools to improve their skills and productivity. He also mentions the importance of mentoring to achieve this goal, taking as an example the legendary IBM CEO Thomas J. Watson, who was *shown how to* sell cash registers by an older, more experienced sales manager.
  However, the concept of *code katas* takes the *showing* approach one step closer to software development. Several books and papers mention the concept, and the studied project was also highly influenced by katas as a teaching device. As an introductory vehicle to the application framework, they were successful, as stated by all interviewees. However, few used them as *deliberate practice*, which was one of the original goals of the katas.
  There is evidence that the teams performing the katas in a group session increased collective learning by making the group discuss individual problems and solutions.

Summary: When teams are developing and testing features in parallel, the importance of having a shared professional culture increases. To keep a coherent architecture, onboarded teams and individuals received structured training, and everyone was expected to contribute to the culture of learning. The shared culture was encouraged by several cross-team forums, and three checklists were used as "Definition of Done" checkpoints, corresponding to the development phases.

All interviewees stated that the code kata exercises were effective in increasing the understanding of the application framework and the expected professional behavior, including testing strategies. However, there is no evidence that participants used the katas to improve their skills beyond the initial try, indicating that the goal of *deliberate practice* was not met.

5.4 F Feedback

Feedback loops have always been important in the software industry, as described both by Royce in 1970 [72] and by Brooks (B1) in 1975 [13]. However, the last 50 years have seen an immense change in *speed* and *automation* of both feedback loops and the software delivery pipeline.

Feedback is one of the five values of the Agile method Extreme Programming (XP) [5], and it is intimately tied to the sprint practice of Scrum [6], which also includes explicit review practices.

Lean Software Development [67] also focuses on feedback. In particular, the practices of *Deliver as fast as possible* and *Build integrity in* highlight the importance of caring for the feedback loops and striving to optimize them, both from a latency and robustness point of view.

Much of the craftsmanship principles detailed in Table 11 are similar to, or complements, Agile or Lean principles, which is acknowledged in several books, for example, as stated by Mancuso in B9 [52]: "Agile methodologies help companies to do *the right thing*…Software Craftsmanship helps developers and companies to do the *thing right*."

F1 On-site customer (proxies)

- **Literature:**
  Books B2, B7, and B8 all mention the importance of close collaboration between the requirement owner and the development team, something that also is a crucial trait of Agile (e.g. [5, 6]) and Lean [67] processes.
  Papers P3 and P5 use the term *Product Owner*, and report that *close collaboration and communication* between the development team and the requirement engineer reduce the waiting time for clarification or re-prioritization of requirements. Paper P7 is cited as the

Table 11. References to ***F*** *Feedback*

Id	Name	Books	Literature	Qualitative
F1	On-site customer (proxies)	B2, B7, B8	P3, P7	Dev1, Dev3, Test1, Test2
F1.1	Requirements	B7, B9	P9	SwArch1, Test2
F1.1.1	Accessible	B2	P3, P9	Dev1, Test2
F1.1.2	Collaborative	B1, B2, B3, B7, B8 B9, B11	P3, P5	Dev1, Test1, Test2
F1.2	Frequent demos	B2, B3, B8, B9, B11		Test1, Test2
F2	Short feedback loops	B2, B3, B4, B6, B7 B8, B9	P1, P3, P4, P5	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
F3	Review	B1, B2, B6, B7, B8	P3	SwArch1, Dev1, Dev2, Test2
F3.1	Team review	B3, B6, B7, B8, B9	P5	SwArch1, Dev2, Dev3, Test1
F3.2	Static review tools	B4, B7	P5	SwArch1
F3.3	Solution review	B7, B9		Dev1, Dev2, Dev3, Test2
F4	Learning from feedback	B2, B3, B6, B7, B8 B9		SwArch1, Dev1, Dev2, Test1, Test2
F5	Continuous integration and tests	B1, B2, B3, B4, B7 B8, B9, B11	P3, P5, P11	SwArch1, Dev2, Test2
F5.1	Frequent release candidates	B1, B2, B3, B9, B11	P5	SwArch1
F5.2	Reproducible releases	B1, B2, B3, B4, B8, B9	P7	

inspiration for the Scrum process [6] and stresses the technical contributions of the Project Manager and Product Manager roles in the studied product.

- **Empirical findings:**
  In the studied case, the requirements were version-controlled and located in a single wiki-based tool since early 2012. Prior to that, requirement engineers were using a proprietary tool, much less accessible. "[referring to old req. tool] — Oh, that was a tool…It took me ages to learn how to upload an Excel file there. We were supposed to tag requirements to test cases. It was terribly unwieldy…But then we got [the new tool]…We could structure it to fit our needs, with requirements as user stories with a version, a history, in one place, reachable for everyone, regardless of whether you are a tester, developer or system tester."(Test2)
  As part of the development phase, teams demoed potential solutions for the proxy customers, who provided feedback and direction.
  "I would say that we talk to the [requirement engineer/proxy customer] at least for half an hour every other day, during the development of a feature. More in the beginning and in the end, and maybe with a more quieter period in the middle. But I would say we talk to them a lot in the middle too…About things that pop up, in code, that maybe are not like the requirement was stated.…"(Test1)
  "…I was just asking the requirements engineer: 'Is it really this, or you wanted something else?'"(Dev1)
- **Analysis:**
  As stated in both the SLR and case study results, software craftsmanship values cooperation rather than confrontation and constant contract negotiation between developers and requirement owners.
  However, constant cooperation also means that requirements need to be in a *single, accessible* and *version-controlled* space, which tracks the evolution of the shared knowledge. This is crucial in order to *know the current status*.

F2 Short feedback loops

- **Literature:**
  Seven of the studied books (B2, B3, B4, B6, B7, B8, and B9) emphasize the importance of getting quick and relevant feedback on all development tasks. Book B6 explicitly states that practice without periodic feedback risks developing bad habits and voices the importance of giving less experienced developers feedback. As stated in item **D1**, book B2 mentions fast feedback as crucial to incremental development, as it allows adjusting direction before it has progressed for too long. Papers P3, P5, and P9 highlight the importance of *fast feedback loops*, also for distributed teams.
- **Empirical findings:**
  In the studied case, product development emphasized getting fast, relevant feedback from customers or internal proxies. There was an urge to slice large requirements into several pieces, each building on the previous, but deliverable and testable on its own.

Table 12. Elapsed calendar days per feature size and activity. **No QA** is the number of features where planned system verification was deemed unnecessary

		Development			**No QA**	**QA Performed**			
Est.size	N	$\hat{x}$	$\bar{x}$	σ	N	N	$\hat{x}$	$\bar{x}$	σ
X-Small	122	22	28.3	24.8	37	85	7	13.2	16.5
Small	109	29	35.2	30.9	24	85	8	18.9	26.2
Medium	72	47.5	61.3	47.3	10	62	16.5	26.5	31.3
Large	13	62	60.4	49.7	1	12	20.5	21.6	10.7

Table 12 shows data from 316 features, whose size was estimated into one of four categories by an estimation group before development started. The table contains the number of features of each size (N), and the median ($\hat{x}$), mean ($\bar{x}$) and standard deviation (σ) of the number of calendar days spent in the development (including design analysis) and system verification (QA) phases. The collected data refers to the period between June 2012 and December 2016. We tested each group with linear regression and found no statistically significant change (either positive or negative) between either the development or the verification duration over the studied period.

The table shows that the organization developed more X-Small (122) and Small (109) features than Medium (72) or Large (13) ones. This suggests that rather than spending months developing several large "chunks of related functions," the organization valued getting feedback, both from system testing organizations and real installations. All four groups have median values lower than mean values, indicating right-skewed distributions.

Features deemed unlikely to impact quality attributes such as performance, stability, or usability were not individually validated in system verification. As indicated in the **No QA** column, this affected 30% of the X-Small and 22% of the Small features. Statistics for features in system verification are shown in the **QA Performed** columns.

Half of the X-Small features spent less than 22 days in development, including design analysis. This is interesting as the organization used three-week sprints, indicating that these features took around one sprint to complete. Examining the commit statistics for these features reveals that the median number of days spent in development (i.e., not considering design and analysis) was 12.5, with a larger mean of 20.8 and a standard deviation of 26.7 days. The system testing organization was also using three-week sprints, which could explain why the larger features were using close to 21 calendar days on average.

As described in item **F5**, teams constantly worked to keep feedback loops from the Continuous Integration builds as short as possible. This involved both utilizing hardware by executing tests in parallel and redesigning test cases (e.g., avoiding sleep statements).

- **Analysis:**
Table 12 indicates that the majority of features were estimated to be X-Small or Small and that this is also reflected in the development and system verification time. However, as indicated in the table, some features are, due to their nature, impossible to slice into smaller parts. This affected 27% of all features, most of them medium-sized. Planned system verification was omitted in 72 of the analyzed features, meaning that more than one in five (22%) features were deemed only to contain functional aspects, which was validated only by the development team before being deployed in production.

F3 Reviews

- **Literature:**
Reviews have long been used as a tool to judge solutions and provide knowledge sharing, and books B2 and B6 state that the review process goes both ways, where junior developers also review everything produced by the team for the purpose of learning. Book B8 recommends pair programming as an efficient and effective form of instant code review, and papers P3 and P5 confirm the importance of frequent reviews as the core of Software Craftsmanship principles.
Two books (B4 and B7) and paper P5 mention the importance of tools that automatically perform some review, including enforcing formatting rules.
Regarding reviews of solution proposals, there are contrary opinions in B7. One interviewee (Brendan Eich) states that this implies a waterfall process, which should be avoided. Still, two other interviewees state that an adequately prepared design review can strengthen the solution. However, they make a distinction between an internal design review, whose purpose is to criticize or find omissions in the implementation, and an external review, involving clients, clarifying that the proposed solutions solve the intended problem.
- **Empirical findings:**
In 2012, following the expansion to the first remote site, the studied organization started using a wiki platform supporting page templates to introduce an Implementation Proposal (IP). For each feature to implement, each team was expected to produce an IP to be reviewed in a team architect (TA) forum. While team architects reviewed the technical solutions, a test responsible also took part in weekly recurring meetings (QA group) focusing on test structure and test strategies; see item **C3**.
During the studied period, 586 IPs were produced, of which 460 were using the wiki-based format (starting from January 2012). Surprisingly, we also found 24 requirements without a corresponding IP. In 4 of these cases, the actual requirement was canceled without being completed. In the remaining 20, there was other reasons for omitting the proposal, such as the solution being described elsewhere or the lead architect doing the implementation himself.
In 34 out of the 460 wiki-based IPs, the first code commit predated the creation of the IP page, and in 15 cases, it happened on the same day. This indicates that teams were prototyping (on a personal or team-based branch) as part of writing the proposed solution. The IP page contained various sections that were actively updated during both the development and the system testing phases.
Related to code reviews, human reviewers should focus on content rather than style. To meet this goal, as described in item **C1**, mandatory code formatting rules and static checks

using the PMD and FindBugs tools were introduced, causing the build to fail in case of violations. An earlier attempt in using advisory Sonar rules (post-commit, sending feedback through email) proved unsuccessful, as most developers ignored these warnings.
The product started using advisory PMD checks in August 2012 and made them mandatory in December 2012. The number of checked rules was initially small but grew over time. At the end of the study, it comprised 373 FindBugs, 155 built-in, and 7 application-specific PMD rules, developed by a team architect to flag particular code patterns as unwanted in the application code.
Starting in April 2012, a number of invariant-checking unit tests, called "metatests," were developed to give fast developer feedback on the expected behavior of the produced code. The meta-tests scanned the project classpath, performing static checks on classes that match particular application-specific criteria. Examples of such tests are "Request and Response classes shall have validation annotations on all fields" and "All remote-invoked methods must have an audit log annotation."
The first Gerrit review took place in June 2013. During the studied period, 3802 reviews took place, out of 54637 total commits. One interviewee indicated that the team used pair programming rather than Gerrit-based reviews: "Our team made a decision not to use Gerrit for review. Instead, we were pairing up, reviewing by sitting close, working on the same task, and interacting with each other's code."(Dev3)

- **Analysis:**
Reviews can be used both to spread knowledge and to enforce an architectural direction. However, to be effective, they require motivated, knowledgeable, and accessible reviewers. As evidenced in the findings, the solution review step did not preclude coding. In over 10% of the found cases, the first line of feature code (presumably a prototypical solution) predated even creating an empty IP page. Instead, the solution review should focus on whether the proposed solution aligns with the overall architecture and direction of the product and sharing the concepts and the approved design between different teams.
However, feedback frequency is also important - it is wasteful to spend effort in a direction not aligned with the overall product architecture. Thus, architects should discuss the intended solution before starting to write a formal implementation proposal.
Static review tools have the advantage that they are objective, consistent, and persistent, but they are limited in scope and have the disadvantage of flagging false positives. The tool can function as a teaching device by tailoring the tool error message or adding application-specific rules. This studied case used the PMD tool to meet this end.

F4 Learning from feedback

- **Literature:**
Six books (B2, B3, B6, B7, B8, and B9) report on the importance of learning from received feedback, with book B6 stating that useful feedback needs to be possible to act upon.
Papers P3, P5, and P11 state the importance of *learning through fast feedback loops* and *ongoing move-testing-experiments*. As discussed in item **C5**, this is also intimately coupled with a culture of learning.
- **Empirical findings:**
Five interviewees mention software development as a learning exercise and highlight reviews as a tool to share knowledge and get feedback, not block development. One interviewee reflects on the importance of learning from customer feedback: "[reacting to defect reports by]…taking a step back, and analyze: 'This was an area that the customers were into…Are there more black spots like that?'"(Test1)

To a large extent, the practices in item **C5**, being focused on learning, also apply here. The team architect and quality assurance roles (TA & QA, see **C3**, **A1**) were also expected to guide their team members via regular feedback and share experiences across teams.

- **Analysis:**
  By focusing on the learning experience of software development and striving to use feedback (whether automated or manual) to learn new and better development practices, it can be argued that the organization as a whole prioritizes learning in a structured way. This is also exemplified by the Lean principle of *Amplify learning* [67].

F5 Continuous integration and tests

- **Literature:**
  As stated by Brooks in his commentary to the 20th anniversary of the original publication of B1, technological progress has led to that "[Microsoft] rebuild the developing system every night [and run the test cases]" [13]. These days, when 25 more years have passed, the nightly runs have been replaced with on-demand-builds, which run after each check-in. The importance of this evolution is stated in eight of the studied books, and papers P3, P5, P9, and P11 also discuss the benefits of *continuous integration and regression testing* for software craftsmanship.
- **Empirical findings:**
  Automated build tools, first Hudson, then Jenkins, were used since the inception, including mandatory testing phases following the compilation and building of the software. The organization relied on personal responsibility, with code signing using personal certificates (see item **C5**), although the release building process was highly automated using build tool plugins, enforcing rules about tagging and versioning of artifacts and dependencies.
  As seen in Figure 5 (see item **D2**), the amount of test code soon eclipsed the amount of production code, as the number of test cases kept growing along with the product functionality. Initially, the test suite was executed sequentially, in a monolithic fashion. Later this was broken down into many parallel tasks, each running towards an isolated system under test (SUT), to decrease feedback latency. The management (booking, releasing, reinstalling) of these systems was handled by an own-developed test-host installation and reservation system, utilizing the SUT to the highest possible degree. At the end of the study, each commit was triggering up to 181 parallel integration test tasks.
  In some circumstances, concurrency issues (e.g., threading) caused tests to fail sporadically (flaky tests). One such example was related to alarm sending and logging. The first naïve solution by individual developers was to add sleep statements into the flaky test case, delaying the test execution by a fixed amount of time. In addition to being wasteful of resources (as the test host was not performing any useful tests, delaying feedback), this also caused additional instability, as the required delay would be dependent on the CPU and network load on the physical machine running the virtual machine under test. After discussing in the TA group (see item **C3**), a senior developer made a special "test helper" using barrier synchronization to solve the instability. Further test helpers solved most causes of instability. The remainder (e.g., due to dependencies on manipulating features in complex third-party software) were relegated to nightly runs when the test environment was less used and more stable.
  Between December 2010 and December 2016, the team made 721 candidate releases of the main product. Of these, 248 turned into sharp releases (where 36 were major feature releases, and the rest was smaller defect corrections). On average, this amounts to 10.0 candidates and 3.4 sharp releases per month. Between March and December 2016, the

continuous integration environment made, on average, 1428.6 builds per month on the master branch (not including feature branch builds).

- **Analysis:**
  Many authors can testify to the utility of Continuous Integration. However, running the tests is not enough; the organization must also act on the feedback provided by the test, including fixing errors, unstable tests, and focusing on keeping the feedback cycle time reasonable. The studied organization strove to shorten the feedback loops for the integration tests to give relevant feedback as soon as possible. Test case structure was also regularly discussed in the QA forum (item **C3** and **C4**).
  Making frequent release candidates and releases means that manual intervention in the release process needs to be kept to a minimum. Still, the organization valued the accountability given by personal code signing of individual artifacts, release candidates, and sharp releases. One benefit of frequent releases is that there is no "big-bang effect" when making the sharp release. By that time, recurrent Continuous Integration jobs should already have verified the constituent components and the functional difference since the last release should be small and manageable.

Summary:
As stated in the introduction, feedback loops have been at the core of software development for at least 50 years. However, the tools and frequency of the feedback have changed over the years. The studied organization not only *used* Continuous Integration practices, but also *worked with* them, striving to *optimize*, and get *faster* feedback.

Similarly, realizing the cost and scarcity of human feedback, the organization strove to utilize *review tools*, such as static code review, invariant-checking unit tests, and web-based review tools such as Gerrit. There was a mandatory design review step to spread knowledge and align directions, but this did not prevent teams from prototyping before describing their first proposed solution.

We also see evidence that in some cases, the agreed process (e.g., reviews, solution descriptions) was not followed. This indicates that the organization tolerated deviations from the process, as long as the perceived benefits of the deviation outweighed the perceived costs (e.g., the lack of competence spread or the risk of lower quality).

6 DISCUSSION AND IMPLICATIONS

6.1 The principles and practices of software craftsmanship — in literature and in our case study (RQ1 and RQ2)

Tables 6, 8, 10, and 11 illustrate the overlaps between the literature and the presented anatomy of craftsmanship. Among the most notable discrepancies and expansions, we consider the following.

A key architectural principle in our anatomy is the **A1** *Participating Software Architects*, i.e., architects need to participate in day-to-day software development. This extends the principles from the literature of passionate, skilled technical leaders who lead empowered teams both practically and concretely. We highlight the decision of **A3.2** *Judicious use of third-party products* as a key practice to follow when setting architectural direction. In addition to functional requirements, quality requirements such as testability and upgradeability must be considered when choosing software components. We note that the architectural direction should be exemplified via concrete, testable **A3.3** *Common application patterns*, rather than comprehensive documentation.

Our results also emphasize that tests should be structured in **D2** *layers*, and every test case should be **D2.1** *stable and independent* to reduce dependencies and enable faster fault isolation and correction. Tests were kept in focus through the principle of **D2.3** *Test-focused Development*, with tests developed close to the production code, using **D2.3.1** *Pairing* and **D2.3.2** *Test-Driven*

Development. We also highlight that the relative lack of comprehensive design documentation was alleviated by having a test base of **D2.4** *expressive tests, with a simple structure*, which also served as **D3** *Design documentation*, together with a collaboratively edited wiki system.

An Agile setting expects teams to be self-organizing, without structure imposed by external forces. However, this freedom should be supported by **C2.2** *Clear roles and responsibilities* and shared **C2.3** *Definition of Done* (DoD) criteria, which help all participants in the organization know what to expect, and when to expect it. This is not to say that external forces have to appoint these roles and check on the DoD, only that the team needs to organize so that the roles are set, and the DoD criteria are fulfilled. To gain trust between different stakeholders and to allow corrective actions, **C4** *Visibility* is essential, including backlog, issues, technical debt, and **C4.2** *Visible status*. Another key practice is **C5** *Accountability*, affecting both transparency and reputation.

Like the agile principles, our vision of craftsmanship also focuses on feedback loops, such as **F1.2** *Frequent demos*. The practice of **F3.3** *Solution review* is highlighted to spread knowledge between teams and to ensure that the proposed solution aligns with the architectural direction. It is important to note that, when needed, the proposed solution should be vetted using prototypes and real test cases before the review takes place. The continuous learning organization values **F4** *Learning from feedback* and sees this as positive. Defect reports can be seen as both good and bad. While reoccurring defects are clearly bad practice, the first occurrence of a particular issue is judged from case to case. Metrics are used accordingly.

6.2 What are the consequences of applying the software craftsmanship principles and practices in real life? (RQ3)

Based on the studied case, we found several examples of how software craftsmanship is embodied in practice and the consequences it brings:

- Developing in a **D2.3** *test-focused* way does allow production code to be refactored and shaped into a clear representation. However, as the product accumulates features, the test codebase will grow faster than the production code, more so for the integration test code than for the unit test code. Therefore, it is important to **D2** *test at several layers* and constantly work with the test code, which is as essential to keep **C2.1** *clean as the production code*. Regarding **A3.4** *refactorings*, the studied organization made on average 16.8% refactoring commits during six years, excluding refactorings made as part of regular features.
- The **D2** *test code* serves two purposes — first, it should verify that the system still behaves as it used to do, and second, it should be **D3.1** *readable as a description* of what the system does. In order to meet these goals, the tests need to be **F5** *frequently executed*, and failures or broken builds need to be quickly **F4** *acted upon*. In some cases, organizational support is needed to enforce these norms, and **C3** *communities of practice* can be used to solve this efficiently.
- There is a trade-off to be made related to verification efficiency and correctly mimicking a deployed system. Solutions to **D2.1** *unstable test cases* can include re-architecting or adding helper functions to make them more stable, increasing testability and trust in the test suite, at the cost of allowing deviations from a production system. As these added functions will not be part of the end-to-end delivery, it is important to keep them **A2.1** *architecturally isolated* from the object under test. Later test phases, such as system testing, should then test the product from a black-box perspective.
- **A1** *Software architects* and **A1.2** *senior developers* play important roles in architectural direction and forming a **C2** *common professional culture*. In the studied case, the creation of a **C** *shared professional culture* was facilitated both by relocating the remote teams to the

primary site for a few months, to learn the product and the development process and by the **C6.2** *structured exercises (katas)* used in order to **C6.3** *teach* newcomers the preferred way of developing new features.

- **F2** *Frequent feedback* is important, both from tools, artifacts, and other stakeholders, such as **F1** *requirement owners*, **F3.1** *other peers*, verification engineers, or target installations.
- All interviewees mention the structured, down-to-earth, practical **C6.2** *kata exercises* as important tools to learn the development process and the preferred way of developing the product, particularly in a group setting. However, there are few indications in the studied case that the katas were used as deliberate practice.
- While the organization advocated and the kata exercises taught **D2.3.2** *Test-Driven Development*, the organization also realized that TDD could be a hard technique to master. Nevertheless, tests and verification were kept in **D2.3** *focus* by keeping the development team responsible for automating functional test cases and keeping the manual test cases to a bare minimum.
- Having a **C1** *common toolchain* and striving for **C6** *mastery* of this toolchain is yet another aspect of a common professional culture. Still, this does not mean that the tools should be static. In the studied case, the organization changed tools several times to be more productive. In some cases, the switch was "abrupt" (e.g., version control and build tools), and in some cases, the switch was "gradual" (e.g., supported IDE). The organization should be prepared to **C6.3** *teach members* the new tools, using guidelines, seminars, and **D2.3.1** *pairing*.

We also found instances where the studied organization fell short of the espoused principles—for instance, regarding **C6.2** *kata exercises* being used solely for new developers, in an individual and isolated setting; a few features being developed without the requested **F3.3** *solution review*; and there were certain teams where **D2.3.2** *pairing* and **C6.3** *mentoring* worked better than in others. In this regard, the software craftsmanship principles and practices can be seen more like guiding lights than absolute truths. However, we still think it is worthwhile to study them more.

6.3 Software Craftsmanship vs. Agile Software Development

Following the organization in paper P5 [51], here we compare, in light of the findings from this study, the principles from the Software Craftsmanship Manifesto with the principles in the Agile Manifesto.

6.3.1 Well-crafted software vs. Working software. Software craftsmanship focuses on well-crafted software, while agile software development promotes delivering software as quickly as possible. Therefore, craftsmanship goes beyond project activities reported as the most frequently used agile practices, e.g., *standup meeting*, *backlog*, *sprint/iterations*, and *sprint planning* [84]. According to the State of Agile Report [19], companies applying agile practices rarely report on practices such as **F5** *Continuous integration*, **D2.3.1** *Pairing*, **D2.2** *Automated testing*, **D2.3.2** *Test-Driven Development*, and **A3.4** *Refactoring*. The results of the SLR, together with the findings of our case study, suggest that craftsmanship focuses on offering agile organizations more down-to-earth, technical practices to improve long term stability and quality, e.g., **A2.1** *Isolated and Layered Architecture* or the use of **A3.1** *Minimalistic Frameworks*.

6.3.2 Steadily adding value vs. Responding to change. Rather than only quickly reacting to changes, craftspeople are expected to also come up with their own improvements, such as **A3.4** *refactorings* or improvements in the overall production (e.g., tools, such as optimizing the **C5** *continuous integration* environment or **D2.2** *automated testing*). This is to make sure that **F5.1** *frequent releases* and **F2** *short feedback loops* prevent degradation of the **A** *architecture*, which would limit the ability to continuously and steadily add value.

A review by Kupiainen et al. [46] indicates that the metric with the strongest influence in Agile and Lean contexts was *velocity*, followed by *effort estimate* and *customer satisfaction*. However, we argue that not only velocity but also clean and bug-free code matters. The same authors report that metric information was broadcast in hallways to motivate people to react faster to problems. Thus, our **C4.2.1** *information radiator* practice was also used to influence behavior here.

6.3.3 Community of professionals vs. Individuals and interactions. Emphasizing the community of professionals over individuals implies that craftspeople would be expected to help each other grow through **C6.3** *mentoring*, constructive feedback, and experience sharing [52].

Our literature and case study results confirm the importance of a **C2** *shared professional culture* and **F** *feedback* as essential themes. Quick **F2** *feedback loops* enable organizations to **D1** *develop incrementally*, concentrating on small deliverables with predictable lead-time. This is crucial for keeping a sustainable pace adding value, and, if needed, to "fail fast." The shared professional culture might impact the ability of the organizations to build up a cross-site sense of belonging and foster the creation of shared ways of working in distributed environments.

The growth of open-source communities and the sponsoring and development of open-source software by commercial vendors can also be viewed as emphasizing software development communities.

6.3.4 Productive partnerships vs. Customer collaboration. While Agile focuses on interactions and collaboration with customers, the craftsmanship approach takes a more long-term, strategic view. For craftspeople, the produced artifacts, knowledge, and learning become part of the organizational knowledge and strengthens the ability to respond and assimilate changes. By being **C5** *accountable* and practicing **C4** *visibility and transparency*, craftsmanship brings a balancing force to customer-focused agile practices.

In the studied case, customer collaboration was implemented through customer proxies and in the "Internal live customer" phase, starting after less than a year of development. This proved successful in sharpening the development teams and spreading knowledge about the product and its environment to integration engineers, which helped smoothen the transition to external customer deployments. After deployment to external customers, the requirement inflow increased, but the organization had already achieved a smooth development process and could keep up with demands without compromising quality.

6.4 Software Craftsmanship vs. Lean Software Development

In this subsection, we compare our anatomy, and the case study results, with the seven principles of Lean Software Development, outlined by Poppendieck & Poppendieck in [67].

- *Eliminate waste* can be seen as a core trait also in Software Craftsmanship. By focusing on the *Steadily adding of value*, and principles that encourage that, a responsible craftsman tries to eliminate waste from any processes or tasks.
- *Amplify learning* also lies at the core of craftsmanship, fostering a **C5** *Culture of learning* via **C6.3** *Mentoring* and **C6.2** *Deliberate practice*, and **F4** *Learning from feedback*.
- *Decide as late as possible* is a way to adjust your design up until the last responsible moment, which is core in **D1** *Incremental development*, where **F1.1.2** *Requirement changes* are seen as a comparative advantage.
- *Deliver as fast as possible* puts value on getting real, actionable **F** *Feedback*, on many levels, both via **F3** *Reviews* and **F5** *Continuous integration and tests*, using **F2** *Short feedback loops*.

- *Empower the team* is also at the core of craftsmanship, where the architecture invites **A1.3** *Empowerment*, and the professional culture values **C4** *Visibility and accountability*.
- *Build integrity in* has a direct parallel in the **D** *Iterative design, development, and verification*, where much of the focus is on layered verification in the **D2** *Testing pyramid*, and that the tests should be **D3.1** *usable and readable as documentation* of a running system.
- *See the whole* is arguably the focus of many craftsmanship principles, both from an **A** *Value-focused architecture* theme to the *Productive partnerships* envisioned in the manifesto.

While there are similarities between the lead architect in the studied product and Poppendieck's chief engineer principle [67], there are also differences. The program planning and budgeting were performed by different roles in the studied case, outside the scope of this paper. The lead software architect focused solely on the software and its structure to enable efficient development of features valued by customers while still meeting the required quality requirements. There were also strategic product managers and system managers dealing with customer requirements and strategic directions for the product, also outside the scope of this paper.

6.5 Returning to the Software Craftsmanship Manifesto

Looking at the manifesto[12] values through the lens of our anatomy, we find the following:

- "As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft." In the first line of the manifesto, the authors explicitly value the **C6** *Culture of learning*, and the **F4** *Learning from feedback*. The need for constant practice also aligns with **A1** *Participating Software Architects*. Although **F3** *Reviews* are not explicitly mentioned, this is one example of a setting enabling experience sharing, either automated through static review tools or manual, via solution or code review.
- "Not only working software but also well-crafted software" as a statement does not define what distinguishes the two classes of software. Our anatomy considers well-crafted software as being composed of **A3** *Clean, minimalistic code*, which is **D1** *incrementally developed*, during constant **A3.4** *Refactoring*. The architecture enables **A2.1** *isolated features, using layers*, and features are developed with **D2** *layered testing* in mind. Functional tests are written by the **D1.2** *team that develops the feature*, so that they are **D3.1** *readable as documentation*.
- "Not only responding to change but also steadily adding value" focuses on the longer-term perspective and the ability to add value to the software in a predictable manner continually. To meet this goal, in addition to the well-craftedness mentioned above, the **A** *architecture* should focus on helping value-creation, making it easy to validate changes through **F5.1** *Frequent release candidates* and through **F5** *Continuous integration*. To keep track of the current state of the product and the project, **C4** *Visibility and transparency* are important, as is the management of **C4.1.1** *Technical debt*.
- "Not only individuals and interactions, but also a community of professionals" emphasizes the community aspect of software development, and many items in the anatomy focus on a **C** *Shared professional culture*. Important aspects of a **C2** *Common culture* include fostering **C2.1** *caring* for your artifacts, having a shared sense of **C2.4** *Pride*, and **C2.2** *Clear roles and responsibilities*. To balance the pride, it is also important to keep **C5** *Accountability* and **C5.1** *Humility*, and craftspeople would do well to manage their **C5.2** *Reputation*.
- "Not only customer collaboration, but also productive partnerships" again focus on the longer-term view, where **C5.2** *Reputation* is at stake. Our anatomy mainly focuses on the requirement formalization's collaborative aspects, using the **F1** *On-site customer* approach and **F1.1.2** *Collaborative requirements* elicitation, by constant communication between the design

[12] http://manifesto.softwarecraftsmanship.org/

team and the requirement owner (customer proxy). Likewise, verification is a collaborative endeavor, where **D1.2** *teams take responsibility for delivering functionally verified features.*

To sum up, our anatomy makes no references to the "lone cowboy programmer" craftsman stereotype mentioned by Boehm in P14 [9]. Instead, it emphasizes the community aspects of modern software development, the importance of mentoring and tutoring newcomers to the field, and the need for constant learning in software development. While there are undoubtedly programmers that prefer solitude and would rather not communicate with others, our anatomy concretizes most of the manifesto ideas, bringing evidence on how some of the craftsmanship principles can work in practice. It also emphasizes the need for senior developers to engage in teaching and mentoring, in addition to behavioral rules to foster a shared culture of learning and professional development.

To be fair, our anatomy does not emphasize the linear progression of apprentice, journeyman, and master outlined by McBreen in B2 [57]. Rather than designating individuals into specific labeled categories, the anatomy emphasizes everyone's responsibility to contribute to a culture of learning, caring for the codebase and the architecture. Naturally, the more senior developers would take a more leading approach, such as in the cross-team forums. Likewise, leading developers were cognizant of the importance of a shared professional culture and used both team relocation and kata exercises to try to instill a common way of working to new project members, regardless of their prior experience.

7 VALIDITY

In this section, we discuss the threats to validity from four different angles: *construct validity*, *internal validity*, *external validity* and *reliability* [91].

Construct Validity deals with whether the studied measures really reflect the constructs that the researcher has in mind and what is stated in the research questions, and the ability of the metrics to informs about the concept [69].

For the qualitative data, construct validity was enhanced by the two additional authors reviewing the flexible interview protocol, making clarifications based on this feedback. We also presented an intermediate version of the anatomy to the studied organization, after analysing the interview data, and received valuable feedback.

Much of the quantitative data comes from Git logs, and using such information to illustrate: i) the proportion of development activities (e.g., feature development or refactoring); ii) the iterative nature of the development; and iii) the usage of layered testing; has some risks that can challenge the reliability of the results.

In particular, when dealing with the proportion of development activities, we analyzed individual commit messages and relied on the organization's strict commit tagging policy. Developers had to tag each individual commit with a code depending on the activities they were carrying out. Only 0.2% of the commits were not properly tagged. We tried to mitigate this threat to construct validity by defining a metric on data that was created with the same objective: to be able to identify the development activities. During the studied period, the organization had no organizational goals associated with this metric (e.g., rewards associated to refactorings or bug fixes). Had such goals been used, this metric would not have been reliable, as developers could have been expected to change behaviour to meet such goals.

For analyzing the adherence to incremental development, we use the evolution of the codebase over time, for the major types of source code. One of the main threats to validity in this case is whether the languages (i.e., Java, XML and Scala) are comparable. As XML is much more verbose than Java, it will grow faster, but the main usage in this analysis is not the growth speed itself, but the fact that they grow together in at sustainable pace. In a non-incremental development scenario, we would expect the production code and the unit test code to grow from the start of the project until the start of the development of integration tests, where these two will suffer a sudden decline in their growth and the focus would move to integration. However in this case the different types of code grow linearly, with slightly different speeds.

Finally, regarding our proposed construct of a testing pyramid and layered testing, we use both the fact that developers state that automated tests were important, and the volume and ratio of test code versus production code. Our proposed metrics (lines of code and the ratio of tests versus production code) say nothing about the quality of said code, but they do illustrate that the different classes of code grew over time, and as the product grew more feature-rich, the amount of different test code grew alongside the production code, although at different speeds. We argue that this shows that in this product, developers took care to layer their tests into different categories of tests and that this behavior was consistent throughout the studied period.

An important aspect to consider when using this data source is the branching pattern and how commits were merged or rebased. In the Git version control system, authors may “squash” commits, perhaps performed by different authors at different times, into one new commit, discarding the constituent commits. This was not an approved practice as the studied organization valued seeing the individual commits as they were written and pushed to the central repository.

Most development took place in a single “master” branch for the duration of the study. Features developed in other branches were eventually introduced into the master branch, typically via the Git rebase function, keeping a linear history by rewriting commits. However, during rewriting, the original author information, including the commit date, is preserved, even if the commits are reordered in the git log. This allows statistics based on Git dates to be reliable data sources, as the commit date reflected when the actual code was changed, not when it was introduced into the master branch.

Internal Validity deals with whether there might be other, non-studied factors that could explain some of the findings.

We used the mixed-methods approach of triangulation to increase internal validity We used Google Scholar to search for papers to form a start set. As we only found 4 relevant papers, we added 5 additional based on experience. This personal bias could threaten internal validity. However, we believe that its impact is minimal after performing four forward and backward snowballing iterations. We have screened 478 references, 782 citations, and 146 books during these iterations. Moreover, Mourão et al. have shown that combining the database search with forward and backward snowballing improves the precision and recall of the literature review [60].

Where possible, we used both quantitative and qualitative data sources. However, there might still be other, non-studied, explaining factors that impact the results. We are aware that the studied development project did not adopt all software craftsmanship principles that we identified in the literature. This remains a threat to internal validity of our work.

External Validity concerns the extent to which it is possible to generalize findings and whether the findings are of interest to people outside of the investigated case.

One of the five misunderstandings about case study research is the inability to generalize from a single case [28]. Following Flyvbjerg, we have focused on analytic generalization rather than statistical generalization by comparing the characteristics of the case to a possible target and presenting case-specific characteristics, as much as confidentiality concerns allowed.

We looked outside the studied case by reviewing other literature for findings or themes to increase external validity.

This buttressing is documented in the Systematic Literature Review section of the paper, and the associated data appear as references throughout the results and analysis sections. However, it must be acknowledged that this buttressing is based on limited empirical evidence. Additionally, the results here are only circumscribed to the analyzed context. More studies in other systems and other organizations are needed to better understand the effect that craftsmanship principles might have on the developed product, the development process, and the organization.

Reliability concerns whether the data and analyses are dependent on the specific researchers, and this is a significant threat to validity for this study, as the first author was part of the studied product development during the whole studied period. To increase reliability, the second and third authors were used in a supporting role, with at least one of them being active participants in all interviews. The first author transcribed all recorded interviews. The transcripts were reviewed by the second and third authors, who separately coded three interviews each, for comparison with the first author's codes, who coded all interviews.

The interviews, conducted between July 2018 and January 2019, used a convenience sample of participants, focusing on including many different aspects, illustrating the concepts and principles used in the development process. Two interviewees were from the outsourced site, and two were women. The lead architect was interviewed separately by the second and third authors, as he had worked closely together with the first author during the studied period.

A threat to reliability is that the interviews took place some years after the actual studied events. In addition to memory errors in the interviewed participants, it also meant that it was hard to reach persons who were part of the product for a shorter time. Thus, the views of such "short-lived" participants may have been different than the interviewees.

We strove to reduce memory errors by seeking additional data in quantitative sources (VCS logs, wikis, requirement tools) using archival analysis whenever possible.

8 CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

Regarding **RQ1**, how Software Craftsmanship has been conceptualized in literature, although the principles have a long history in grey literature, we found comparatively few published research articles. In our systematic literature review, we could find only 18 papers discussing the principles to some extent, see Table 4. Based on these papers, we found 11 books, of which seven were new to us before starting this study.

In order to conceptualize the findings, and to illustrate which of these principles and practices that we can see in our studied case (**RQ2**), we drew the anatomy map, comprising of four key themes, with 17 principles and 47 practices; see Figure 3 and Table 6, 8, 10 and 11.

In answering **RQ3**, what consequences applying the practices bring, we drew examples from our studied case, using both quantitative and qualitative data. Most of these principles align well with core Agile and Lean principles but place a higher weight on the technical practices.

Although the Agile and Lean principles seem quite well-researched, the Software Craftsmanship principles seem to warrant more systematic studies by the research community.

8.2 Future Work

This study was performed in a particular setting, having quick feedback cycles from customers with rapidly changing requirements. Whether the principles still apply in other settings, such as in situations with more static and stable requirements, or different organizations, remains to be seen.

In future studies, we intend to study how these practices have affected the defect statistics, internal and external quality, and how the principles have been applied as the organization has changed. We also plan to explore the relationships between Agile and Lean software development and software craftsmanship. We are aware that both Agile and Lean software development have aspects similar and overlapping with software craftsmanship. Thus, we would like to explore this in detail in subsequent publications.

ACKNOWLEDGMENTS

This research was supported by the KKS PLEng 2.0 grant at Blekinge University of Technology, and Ericsson AB, through the SHADE KKS Hög project with ref: 20170176, and through the KKS SERT Research Profile with ref. 2018010 project both at Blekinge Institute of Technology, SERL Sweden.

REFERENCES

[1] D. Šmite, N. B. Moe, G. Levinta, and M. Floryan. 2019. Spotify Guilds: How to Succeed With Knowledge Sharing in Large-Scale Agile Organizations. *IEEE Software* 36, 2 (2019), 51–57. https://doi.org/10.1109/MS.2018.2886178

[2] Pekka Abrahamsson and Juha Koskela. 2004. Extreme programming: A survey of empirical data from a controlled case study. *Proceedings - 2004 International Symposium on Empirical Software Engineering, ISESE 2004* (2004), 73–82. https://doi.org/10.1109/ISESE.2004.1334895

[3] Mohammed Ibrahim Alhojailan. 2012. Thematic Analysis : A Critical Review of Its Process and Evaluation. *WEI International European AcademicConference Proceedings* 1, 2011 (2012), 8–21.

[4] Len Bass, Paul Clements, and Rick Kazman. 2013. *Software Architecture in Practice.* Pearson.

[5] Kent Beck and Erich Gamma. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.

[6] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. 1999. SCRUM: An extension pattern language for hyperproductive software development. *Pattern languages of program design* 4 (1999), 637–651.

[7] Ilias Bergström and Alan F. Blackwell. 2016. The practices of programming. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, Vol. 2016-Novem. 190–198. https://doi.org/10.1109/VLHCC.2016.7739684

[8] Elizabeth Bjarnason, Michael Unterkalmsteiner, Markus Borg, and Emelie Engström. 2016. A multi-case study of agile requirements engineering and the use of test cases as requirements. *Information and Software Technology* 77 (2016), 61–79.

[9] Barry Boehm. 2006. A View of 20th and 21st Century Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. ACM, New York, NY, USA, 12–29. https://doi.org/10.1145/1134285.1134288

[10] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (jan 2006), 77–101. https://doi.org/10.1191/1478088706qp063oa arXiv:1011.1669

[11] Virginia Braun and Victoria Clarke. 2014. What can "thematic analysis" offer health and wellbeing researchers? *International Journal of Qualitative Studies on Health and Well-being* 9 (2014), 20–22. https://doi.org/10.3402/qhw.v9.26152

[12] Ricardo Britto, Darja Šmite, and Lars-Ola Damm. 2016. Software Architects in Large-Scale Distributed Projects: An Ericsson Case Study. *IEEE Software* 33, 6 (nov 2016), 48–55. https://doi.org/10.1109/MS.2016.146

[13] Frederick P Brooks. 1995. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Pearson Education. https://books.google.se/books?id=Yq35BY5Fk3gC

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture. A System of Patterns.* Vol. Vol. 1. John Wiley & Sons, Chichester. 476 pages.

[15] Oisín Cawley, Xiaofeng Wang, and Ita Richardson. 2010. Lean/agile software development methodologies in regulated environments - State of the art. In *International Conference on Lean Enterprise Software and Systems*, Vol. 65 LNBIP. Springer Verlag, 31–36. https://doi.org/10.1007/978-3-642-16416-3_4

[16] Panagiota Chatzipetrou, Darja Šmite, and Rini van Solingen. 2018. When and Who Leaves Matters: Emerging Results from an Empirical Study of Employee Turnover. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 53, 4 pages. https://doi.org/10.1145/3239235.3267431

[17] Paul C Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Meson, Robert Nord, Judith Stafford, Paulo Merson, Robert Nord, and Judith Stafford. 2010. *Documenting software architectures: views and beyond* (2nd editio ed.). Pearson Education. http://dl.acm.org/citation.cfm?id=599933

[18] Mike Cohn. 2010. *Succeeding with agile: software development using Scrum.* Pearson Education.

[19] CollabNet VersionOne. 2019. *The 13th annual STATE OF AGILE Report - 2018.* Technical Report. 16 pages. https://www.stateofagile.com/{#}ufh-i-521251909-13th-annual-state-of-agile-report

[20] James O. Coplien. 1994. Borland Software Craftsmanship: A New Look at Process, Quality and Productivity. In *Proceedings of the 5th Annual Borland International Conference* (Orlando, FL, USA).

[21] Lisa Crispin and Janet Gregory. 2008. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Pearson Education.
[22] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1992), 29–30.
[23] Bill Curtis, Herb Krasner, and Neil Iscoe. 1988. A Field Study of the Software Design Process for Large Systems. *Commun. ACM* 31, 11 (Nov. 1988), 1268–1287. https://doi.org/10.1145/50087.50089
[24] Philipp Diebold and Marc Dahlem. 2014. Agile practices in practice - A mapping study. In *18th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery. https://doi.org/10.1145/2601248.2601254
[25] Edsger W Dijkstra. 1982. On the role of scientific thought. In *Selected writings on computing: a personal perspective*. Springer, 60–66.
[26] Tomaž Dogša and David Batič. 2011. The effectiveness of test-driven development: An industrial case study. *Software Quality Journal* 19, 4 (2011), 643–661. https://doi.org/10.1007/s11219-011-9130-2
[27] Markus Feyh and Kai Petersen. 2013. Lean software development measures and indicators - A systematic mapping study. In *Lecture Notes in Business Information Processing*, Vol. 167. Springer Verlag, 32–47. https://doi.org/10.1007/978-3-642-44930-7_3
[28] Bent Flyvbjerg. 2006. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry* 12, 2 (2006), 219–245. https://doi.org/10.1177/1077800405284363
[29] Martin Fowler, Kent Beck, John Brant, and William Opdyke. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
[30] Davide Fucci, Hakan Erdogmus, and Burak Turhan. 2015. A Dissection of Test-Driven Development: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering* 6, 1 (2015), 1–20. https://doi.org/10.1109/TSE.2016.2616877 arXiv:1611.05994
[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education. https://books.google.se/books?id=6oHuKQe3TjQC
[32] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106, May 2018 (2019), 101–121. https://doi.org/10.1016/j.infsof.2018.09.006 arXiv:1707.02553
[33] T.F. Gilbert. 1978. *Human Competence: Engineering Worthy Performance*. McGraw-Hill. https://books.google.se/books?id=goNbAAAAMAAJ
[34] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company.
[35] Corey Haines. 2014. *Understanding the 4 Rules of Simple Design*. Leanpub. 88 pages.
[36] Rashina Hoda, Norsaremah Salleh, John Grundy, and Hui Mien Tee. 2017. Systematic literature reviews in agile software development: A tertiary study. *Information and Software Technology* 85 (2017), 60–70. https://doi.org/10.1016/j.infsof.2017.01.007
[37] D Hoover and A Oshineye. 2009. *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman*. O'Reilly Media. https://books.google.se/books?id=I3xFAoZT%5C_5AC
[38] Andy Hunt and Dave Thomas. 1999. *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education.
[39] Martin Ivarsson and Tony Gorschek. 2011. A method for evaluating rigor and industrial relevance of technology evaluations. *Empirical Software Engineering* 16, 3 (2011), 365–395.
[40] Ivar Jacobson and Ed Seidewitz. 2014. A New Software Engineering. *Queue* 12, 10, Article 30 (Oct. 2014), 9 pages. https://doi.org/10.1145/2685690.2693160
[41] Samireh Jalali and Claes Wohlin. 2012. Global software engineering and agile practices: A systematic review. *Journal of software: Evolution and Process* 24, 6 (2012), 643–659.
[42] Daniel Karlström and Per Runeson. 2005. Combining Agile Methods with Stage-Gate Project Management. *IEEE Software* May/June (2005), 43–49.
[43] George L Kelling, James Q Wilson, et al. 1982. Broken windows. *Atlantic monthly* 249, 3 (1982), 29–38.
[44] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press, New York, New York, USA, 1. https://doi.org/10.1145/2393596.2393655
[45] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.
[46] Eetu Kupiainen, Mika V. Mäntylä, and Juha Itkonen. 2015. Using metrics in Agile and Lean software development - A systematic literature review of industrial studies. https://doi.org/10.1016/j.infsof.2015.02.005
[47] C Larman and B Vodde. 2008. *Scaling Lean and Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*. Pearson Education. https://books.google.se/books?id=HbRo4kYnTnMC
[48] Rikard Lindell. 2012. The Craft of Programming Interaction. In *Proceedings of International Workshop on the Interplay between User Experience Evaluation and Software Development (I-UxSED 2012)*. 26–30.

[49] Rikard Lindell. 2014. Crafting Interaction: The Epistemology of Modern Programming. *Personal Ubiquitous Comput.* 18, 3 (March 2014), 613–624. https://doi.org/10.1007/s00779-013-0687-6

[50] Jessica Lingel and Tim Regan. 2014. "It's in Your Spinal Cord, It's in Your Fingertips": Practices of Tools and Craft in Building Software. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing* (Baltimore, Maryland, USA) *(CSCW '14)*. ACM, New York, NY, USA, 295–304. https://doi.org/10.1145/2531602.2531614

[51] Percival Lucena and Leonardo P. Tizzei. 2016. Applying Software Craftsmanship Practices to a Scrum Project: an Experience Report. *CoRR* abs/1611.05789 (2016). arXiv:1611.05789 http://arxiv.org/abs/1611.05789

[52] Sandro Mancuso. 2014. *The Software Craftsman: Professionalism, Pragmatism, Pride.* Pearson Education. https://books.google.se/books?id=JxvVBQAAQBAJ

[53] G. Marcionetti, F. Cannizzo, and P. Moser. 2008. The Toolbox of a Successful Software Craftsman. In *Engineering of Computer-Based Systems, IEEE International Conference on the(ECBS)*, Vol. 00. 389–397. https://doi.org/10.1109/ECBS.2008.48

[54] Robert C Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education. https://books.google.se/books?id=_i6bDeoCQzsC

[55] Robert C Martin. 2011. *The clean coder: a code of conduct for professional programmers.* Pearson Education.

[56] Robert C Martin. 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson Education.

[57] Pete McBreen. 2002. *Software Craftsmanship: The New Imperative.* Addison-Wesley.

[58] Jim McCarthy. 1995. *Dynamics of software development.* Vol. 3. Microsoft Press Redmond, WA.

[59] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.

[60] Erica Mourão, João Felipe Pimentel, Leonardo Murta, Marcos Kalinowski, Emilia Mendes, and Claes Wohlin. 2020. On the performance of hybrid search strategies for systematic literature reviews in software engineering. *Information and Software Technology* 123 (2020), 106294.

[61] Hussan Munir, Misagh Moayyed, and Kai Petersen. 2014. Considering rigor and relevance when evaluating test driven development: A systematic review. (2014). https://doi.org/10.1016/j.infsof.2014.01.002

[62] Maria Paasivaara and Casper Lassenius. 2014. Communities of practice in a large distributed agile software development organization – Case Ericsson. *Information and Software Technology* 56, 12 (2014), 1556 – 1577. https://doi.org/10.1016/j.infsof.2014.06.008 Special issue: Human Factors in Software Development.

[63] D. Parsons, A. Mathrani, T. Susnjak, and A. Leist. 2014. Coderetreats: Reflective Practice and the Game of Life. *IEEE Software* 31, 4 (July 2014), 58–64. https://doi.org/10.1109/MS.2014.25

[64] David Parsons, Teo Susnjak, and Anuradha Mathrani. 2016. Design from detail: Analyzing data from a global day of coderetreat. *Information and Software Technology* 75 (2016), 39 – 55. https://doi.org/10.1016/j.infsof.2016.03.005

[65] Kai Petersen. 2012. A palette of lean indicators to detect waste in software maintenance: A case study. In *Lecture Notes in Business Information Processing*, Vol. 111 LNBIP. Springer Verlag, 108–122. https://doi.org/10.1007/978-3-642-30350-0_8

[66] Kai Petersen and Claes Wohlin. 2011. Measuring the flow in lean software development. *Software - Practice and Experience* 41, 9 (2011), 975–996. https://doi.org/10.1002/spe.975

[67] Mary Poppendieck and Tom Poppendieck. 2003. *Lean Software Development: An Agile Toolkit.* Addison-Wesley.

[68] Bill Pyritz. 2003. Craftsmanship versus engineering: Computer programming - An art or a science? *Bell Labs Technical Journal* 8, 3 (2003), 101–104. https://doi.org/10.1002/bltj.10079 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/bltj.10079

[69] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand) *(EASE'18)*. Association for Computing Machinery, New York, NY, USA, 13–23. https://doi.org/10.1145/3210459.3210461

[70] Mark Richards. 2015. *Software Architecture Patterns.* O'Reilly Media, Inc.

[71] Pilar Rodríguez, Kirsi Mikkonen, Pasi Kuvaja, Markku Oivo, and Juan Garbajosa. 2013. Building Lean Thinking in a Telecom Software Development Organization: Strengths and Challenges. In *Proceedings of the 2013 International Conference on Software and System Process* (San Francisco, CA, USA) *(ICSSP 2013)*. ACM, New York, NY, USA, 98–107. https://doi.org/10.1145/2486046.2486064

[72] Winston W. Royce. 1970. Managing the development of large software systems. In *Proceedings, IEEE WESCON*. 1–9. https://doi.org/10.1016/0378-4754(91)90107-E

[73] Per Runeson, Martin Höst, Austen Rainer, and Bjorn Regnell. 2012. *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons.

[74] Dina Salah, Richard F. Paige, and Paul Cairns. 2014. A systematic literature review for Agile development processes and user centred design integration. In *18th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, London, UK. https://doi.org/10.1145/2601248.2601276

[75] Johnny Saldana. 2015. *Coding Manual for Qualitative Researchers* (3rd ed.). Sage Publications. 223 pages.

[76] T. Sedano. 2012. Towards Teaching Software Craftsmanship. In *2012 IEEE 25th Conference on Software Engineering Education and Training*. 95–99. https://doi.org/10.1109/CSEET.2012.29

[77] P Seibel. 2009. *Coders at Work: Reflections on the Craft of Programming*. Apress. https://books.google.se/books?id=2kMIqdfyT8kC

[78] R Sennett. 2008. *The Craftsman*. Yale University Press. https://books.google.se/books?id=SRAV0KvuOQwC

[79] Ana Silva, Thalles Araújo, João Nunes, Mirko Perkusich, Ednaldo Dilorenzo, Hyggo Almeida, and Angelo Perkusich. 2017. A Systematic Review on the Use of Definition of Done on Agile Software Development Projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering* (Karlskrona, Sweden) *(EASE'17)*. Association for Computing Machinery, New York, NY, USA, 364–373. https://doi.org/10.1145/3084226.3084262

[80] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. (2016). https://doi.org/10.1145/2884781.2884833

[81] Paul Taylor. 2003. Vernacularism in Software Design Practice: does craftsmanship have a place in software engineering? *Australasian Journal of Information Systems* 11, 1 (2003). https://doi.org/10.3127/ajis.v11i1.143

[82] Dave Thomas. 2010. Professional Developers Practice their Kata to Stay Sharp. *Journal of Object Technology* 9 (03 2010), 23–25. https://doi.org/10.5381/jot.2010.9.2.c3

[83] Ayse Tosun, Oscar Dieste, Davide Fucci, Sira Vegas, Burak Turhan, Hakan Erdogmus, Adrian Santos, Markku Oivo, Kimmo Toro, Janne Jarvinen, and Natalia Juristo. 2017. An industry experiment on the effects of test-driven development on external quality and productivity. *Empirical Software Engineering* 22, 6 (dec 2017), 2763–2805. https://doi.org/10.1007/s10664-016-9490-0

[84] Raoul Vallon, Bernardo José da Silva Estácio, Rafael Prikladnicki, and Thomas Grechenig. 2018. Systematic literature review on agile practices in global software development. *Information and Software Technology* 96, April 2017 (2018), 161–180. https://doi.org/10.1016/j.infsof.2017.12.004

[85] Darja Šmite, Nils Brede Moe, Marcin Floryan, Georgiana Levinta, and Panagiota Chatzipetrou. 2020. Spotify Guilds. *Commun. ACM* 63, 3 (Feb. 2020), 56–61. https://doi.org/10.1145/3343146

[86] Etienne Wenger. 1999. *Communities of practice: Learning, meaning, and identity.* Cambridge university press.

[87] Ron Westrum. 2004. A typology of organisational cultures. *BMJ Quality & Safety* 13, suppl 2 (2004), ii22–ii27.

[88] R Winter. 2015. *Agile Performance Improvement: The New Synergy of Agile and Human Performance Technology*. Apress. https://books.google.se/books?id=t2QnCgAAQBAJ

[89] Niklaus Wirth. 2008. A Brief History of Software Engineering. *IEEE Annals of the History of Computing* 30, 3 (2008), 32–39. https://doi.org/10.1109/MAHC.2008.33

[90] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. ACM, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[91] R.K. Yin. 2009. *Case Study Research: Design and Methods*. SAGE Publications.

[92] Ehsan Zabardast, Javier Gonzalez-Huerta, and Darja Šmite. 2020. Refactoring , Bug Fixing , and New Development Effect on Technical Debt : An Industrial Case Study. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 376–384. https://doi.org/10.1109/SEAA51224.2020.00068