Towards Understanding Software Craftsmanship

Anders Sundelin

Blekinge Institute of Technology Licentiate Dissertation Series
No 2021:07

Towards Understanding Software Craftsmanship

Anders Sundelin

Licentiate Dissertation in
Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
SWEDEN

2021 Anders Sundelin
Department of Software Engineering
Publisher: Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
Printed by Exakta Group, Sweden, 2021
ISBN: 978-91-7295-427-4
ISSN: 1650-2140
urn:nbn:se:bth-22041

"Those who cannot remember the past are condemned to repeat it."
- George Santayana; The Life of Reason, 1905

Abstract

The concept of software craftsmanship has roots in the earliest days of computing but has received comparably little attention from the research community. As a reaction to how Agile methods were practiced and taught in industry, in 2009, the Manifesto for Software Craftsmanship was formulated and published, drawing attention to the concept. Subsequent books and research papers have also elaborated on the concept.

With this dissertation, we aim to study the software craftsmanship phenomenon using empirical software engineering methods. We developed an anatomy of software craftsmanship through a systematic literature study and a longitudinal case study, following a project consisting of multiple teams over several years. We also illustrate some consequences of not following through on the espoused craftsmanship practice of managing and account for technical debt. We find that some areas exhibited high growth in technical debt, while others remained comparably idle. This indicates that it is important to keep track of existing technical debt, but repayment should consider the distribution of each kind of technical debt in the codebase.

Our studies are empirical, using mixed methods, analyzing quantitative as well as qualitative data. We used thematic coding to structure the qualitative data into themes, principles, and practices.

We provide our systematically derived anatomy of the principles and practices of software craftsmanship and discuss how these relate to other principles within software engineering in general.

Keywords: *Software Craftsmanship, Empirical Software Engineering, Agile Software Development*

Acknowledgements

I am deeply indebted to my family—especially Katti, my dear wife, who has borne the brunt of the work when I'm off into the clouds or deep into the literature.

My supervisors and colleagues at Blekinge Institute of Technology have always been patient and supportive, striving to increase my understanding of the essence of being a researcher.

Finally, my colleagues, past and present, at Ericsson AB—without you, this research would have been for naught. Thank you all for your support and interest in being part of this journey.

Overview of Papers

Papers in this Thesis

- **Chapter 2: Anders Sundelin**, Javier Gonzalez-Huerta, Krzysztof Wnuk. "Test-Driving FinTech Product Development: An Experience Report" Conference proceedings *PROFES 2018 - Product-Focused Software Process Improvement*, 2018. DOI: 10.1007/978-3-030-03673-7_16

- **Chapter 3: Anders Sundelin**, Javier Gonzalez-Huerta, Krzysztof Wnuk, Tony Gorschek. "Towards an Anatomy of Software Craftsmanship" *ACM Transactions on Software Engineering and Methodology (TOSEM)*. Accepted for publication 2021-05-27. DOI: 10.1145/3468504

- **Chapter 4: Anders Sundelin**, Javier Gonzalez-Huerta, Krzysztof Wnuk. "The Hidden Cost of Backward Compatibility: When Deprecation Turns into Technical Debt - An Experience Report," in *Proceedings of the 3rd International Conference on Technical Debt, TechDebt '20*, 2020. DOI: 10.1145/3387906.3388629

- **Chapter 5: Anders Sundelin**, Javier Gonzalez-Huerta, Krzysztof Wnuk, Tony Gorschek. "Dear Lone Cowboy Programmer - your days are numbered!" Submitted 2021-07-22 to *Communications of the ACM*. Under review.

Contribution Statement

Anders Sundelin is the lead author of all the papers in this thesis. As a lead author, he took the main responsibility in designing the studies, collecting and

analyzing data, and reporting the findings in peer-reviewed publications. Furthermore, he is the sole author of Chapter 1, the overview. The co-authors' contributions are described below.

Chapter 2: Krzysztof Wnuk wrote parts of Chapter 2.1, reviewed and commented on intermediate versions and the final draft of the paper. Javier Gonzalez-Huerta reviewed and commented on intermediate versions and the final draft of the paper.

Chapter 3: Both Javier Gonzalez-Huerta and Krzysztof Wnuk participated as researchers, reviewed, and commented on intermediate versions of the paper. Krzysztof Wnuk summarized the SLR paper findings in Table 3.4 and was, together with Javier Gonzalez-Huerta, highly influential in the methodology section. Tony Gorschek provided valuable review insights and comments on several versions of the paper, including how to structure the large paper into digestible parts.

Chapter 4: Javier Gonzalez-Huerta provided valuable support regarding how to present the different classes of TechDebt-items and reviewed and commented on all drafts of the paper. Krzysztof Wnuk reviewed and commented on intermediate versions and the final draft of the paper.

Chapter 5: Javier Gonzalez-Huerta wrote parts of Chapter 5.2.2, reviewed and commented on intermediate versions and the final draft of the paper. Krzysztof Wnuk wrote parts of Chapter 5.1, reviewed and commented on intermediate versions and the final draft of the paper. Tony Gorschek reviewed and commented on intermediate versions and the final draft of the paper.

Funding

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.

Contents

Abstract **vii**

Acknowledgements **ix**

Overview of Publications **xi**

Papers in this Thesis xi

List of Abbreviations **xv**

1 Overview **1**

1.1 Introduction 1
1.2 Background and Related Work 3
1.3 Research Questions and Contributions 4
1.4 Methodology 7
1.5 Conclusion and Future Research 12

2 Test-Driving FinTech Product Development **15**

2.1 Introduction 15
2.2 Background and Related Work 16
2.3 Case Description and Analysis Method 17
2.4 Results and Discussion 18
2.5 Implications for Research and practice 23

3 Towards an Anatomy of Software Craftsmanship **25**

3.1 Introduction 26
3.2 Background and Related Work 27
3.3 Research Methodology 28
3.4 Systematic Literature Review Results 37

3.5 The Anatomy of Software Craftsmanship 41
3.6 Discussion and Implications 82
3.7 Validity 90
3.8 Conclusions and Future Work 93

4 The Hidden Cost of Backward Compatibility: When Deprecation Turns into Technical Debt 95
4.1 Introduction 96
4.2 Related Work 97
4.3 Research Methodology 99
4.4 Results 104
4.5 Threats to validity 120
4.6 Conclusions 122

5 Dear Lone Cowboy Programmer - your days are numbered! 123
5.1 Introduction 124
5.2 Main observations 127
5.3 Conclusions 135

References 137

List of Abbreviations

Abbreviation	Definition
API	Application Programming Interface
ATDD	Acceptance-Test-Driven Development, a development methodology
BDD	Behaviour-Driven Design, a development methodology
CoP	Community of Practice (also: Communities of Practice)
CPU	Central Processing Unit
DDD	Domain-Driven Design, a development style
DoD	Definition of Done
DSL	Domain-Specific Language
EJB	Enterprise JavaBeans, a Java standard
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Implementation Proposal
IQR	Inter-Quartile Range
ITLD	Iterative Test-Last Development, a development methodology
kLOC	kilo-LOC, one thousand lines of code
LOC	one line of (non-commented) program code
PMD	A tool used for static code analysis
PO	Product Owner, a role
QA	Quality Assurance, a process step, or a role
SLR	Systematic Literature Review
SUT	System Under Test
TA	Team Architect, a role
TDD	Test-Driven Development
TD	Technical Debt
UML	Unified Modeling Language
VCS	Version Control System
XML	Extensible Markup Language
XP	Extreme Programming, a development methodology
XSD	Extensible Stylesheets, used for processing XML

List of Figures

1.1 Research Methodology Overview. 8

2.1 Lines of code for each category, per release, P01, P02 and P03 are initial prereleases and R01 is the first commercial release. . . 20
2.2 Number and ratio of corrected defects for different versions. . . . 21

3.1 Timeline of major events in the studied system. 31
3.2 Process for Building the Anatomy of Software Craftsmanship. . . 35
3.3 The anatomy of Software Craftsmanship. 42
3.4 Layered view of the Initial architecture (a) and Layered view of the Architecture after separating protocols from business logic (b). 46
3.5 Ratio of test code vs. production code over time. 55

4.1 Schematic view of the studied system, exposing services in different versions. 101
4.2 Production and test files per quarter 102
4.3 Occurrences of services in Java or integration test code 107
4.4 Top 6 used deprecated service versions per quarter 109
4.5 Top 7-12 used deprecated service versions per quarter 110
4.6 Top 6 used deprecated service versions in Java files 111
4.7 Percentage of identical integration test files and LOC 114
4.8 Schematic illustration of Principal, Interest, First and Last metric for integration tests of service Alpha v1.0 115
4.9 Count of commits affecting files using the top six deprecated service versions across each quarter 118

List of Tables

2.1 Mean and median number of lines per file. 19

3.1 Snowballing Iteration Statistics and Results 31
3.2 Quarterly Developer Statistics 33
3.3 Case Study Interviewee Background, Ordered by Industry Experience 34
3.4 Papers Resulting from the Systematic Literature Review 38
3.5 Books Resulting from the SLR. Boldface Books Influenced Initial Anatomy. 40
3.6 References to ***A*** *Value-focused architecture* 43
3.7 Summary Statistics of the Proportion and Type of Main Branch Commits per Quarter 50
3.8 References to ***D*** *Iterative design, development, and verification* . 51
3.9 Summary Code Statistic for the Five Major Code Types 52
3.10 References to ***C*** *Shared professional culture* 61
3.11 References to ***F*** *Feedback* 74
3.12 Elapsed Calendar Days Per Feature Size and Activity. 76

4.1 Average lines per file across quarters. 103
4.2 Average number of authors each quarter, per year 104
4.3 Occurrences in files. 108
4.4 Most growing deprecated service versions, per quarter. 112
4.5 Least growing deprecated services, per quarter. 113
4.6 Affected files for all deprecated services 113
4.7 Affected files for unused services (Type-I TD items). 116

5.1 Elapsed Calendar Days Per Feature Size and Activity. 129

Chapter 1

Overview

1.1 Introduction

The history of computing has seen computers evolve from unique, specialized, state-funded machines operated by strictly guarded principled programmers to the ubiquitous computers, both embedded and general-purpose, that nearly every person on the planet interacts with daily [13], [139]. This exponential increase in computing power and availability has resulted in an explosion of software professionals, from the few principled "program operators" of the 1950s to today, where the market research firm Statista[1] estimate that the number of active software developers will grow from nearly 24 million in 2019 to more than 27 million in 2023. Furthermore, today an interested and motivated non-professional developer can build advanced applications using ever more capable (often gratis) tools, such as spreadsheets, databases, web-frameworks, and even model-building tools in Artificial Intelligence and Machine Learning. This suggests that the number of people solving problems by constructing new software is likely many times higher.

The ever-changing software technology requires constant software development skills update. Languages, tools, and frameworks evolve and are updated as features are added or vulnerabilities are fixed. The failure to update deployed software can lead to breaches and loss of a substantial amount of money

[1] https://www.statista.com/statistics/627312/worldwide-developer-population/

and goodwill, as evidenced by the failure of Equifax to patch a two-month-old vulnerability in the open-source Apache Struts framework[2].

As evidenced in the Systematic Literature Review part of this thesis, Section 3.4, the concept of Software Craftsmanship has a long history in computing, dating from the earliest days of programming. Despite this, the number of peer-reviewed papers written about craftsmanship remains low. Books dominate the field, with McBreen [91] advocating the concept, contrasting it with a waterfall-model of Software Engineering, similar to the way Royce [113] in 1970 set up the model inspiring the original waterfall model of software development in order to advocate an iterative, feedback-driven model (which, sadly, was largely ignored for the next 20 years).

Martin [87] popularized the term by putting it in the subtitle of his book that served as an initial inspiration for the case study described later in this thesis. The Manifesto for Software Craftsmanship[3] was published in 2009, spurring further development and more books. The general motivation for the manifesto was the perception that Agile adoption in the industry was ignoring the more technical Agile methodologies, such as Extreme Programming (XP), and was focusing too much on the commercialization of more process-oriented methodologies such as Scrum [83].

The motivation for the work presented in this thesis is to understand the concept of Software Craftsmanship and how it relates to current software engineering practices. This thesis is structured based on four articles, three of which have been published in peer-reviewed publications, and the fourth submitted to a peer-reviewed venue.

The rest of this chapter discusses the general theme of software craftsmanship, with Section 1.2 discussing the overall background and related work. Section 1.3 states the overall research questions and the research contribution, while Section 1.4 discusses the methodology. The chapter closes with Section 1.5, drawing conclusions and highlighting future research avenues.

The remainder of the thesis includes chapters published in, and submitted to, peer-reviewed publications, as detailed in each chapter. Chapter 2 describes the general context and draws preliminary conclusions based on mainly quantitative metrics from a part of the studied system. Chapter 3 combines qualitative analysis of literature findings and interview data with quantitative analysis from a larger part of the studied system to draw a general concept map of themes, principles, and practices. Chapter 4 discusses some of the consequences, in par-

[2]https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/

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

ticular, deprecation and backward compatibility requirements of applying these practices to the test base of the studied system. Finally, Chapter 5 highlights the findings and provides additional data bolstering our case for craftsmanship.

1.2 Background and Related Work

Books dominate the prior works on Software Craftsmanship, although we also found a few research articles, as evidenced in Chapter 3.4. The concept was mentioned by Brooks [17] nearly 50 years ago. In 2002, McBreen [91] contrasted the Software Craftsmanship approach with the classical definition of Software Engineering, citing the IEEE 1990 definition of Software Engineering, which he claimed was a useful process for life-critical applications, but not when money or budgets were a constraining factor.

In his seminal work defining the principles and importance of clean code, Martin [87] included the term in the sub-title, which spread the term to a broader audience. In 2014, Mancuso [83] wrote a book elaborating the concept even further. Meanwhile, the interest in the concept had spurred conferences and user groups worldwide, such as in the US[4], UK[5], and Germany[6].

However, these books paint a picture of contemporary software engineering that—while perhaps common in industry state-of-practice—fails to represent the contemporary view of software engineering in research, particularly research into Behavioural Software Engineering [77]. Starting from the closing argument in McBreen: "Software development is meant to be fun. If it isn't, the process is wrong", there is a direct connection to the research done on the happiness of software developers [54], [55], and its impact on the performance of the individual, team, and organization.

In the chapter on Craftsmanship in Martin's memoir [85], Mancuso states that craftsmanship should be seen as an ideology, not a methodology. As such, craftsmanship does not have direct practices but instead perpetuates the search for better practices and ways of working. He also states that an essential part of craftsmanship is professionalism and enabling clients to achieve their business goals; it is wrong to assume that craftsmanship solely focuses on technical practices, engineering, and self-improvement.

Test-Driven Development (TDD) is one of the most known practices of Agile processes and is particularly associated with Extreme Programming (XP). How-

[4]https://scna.softwarecraftsmanship.org/
[5]https://sc-london.com/
[6]https://www.socrates-conference.de/

ever, researchers have identified several barriers to its wider industrial adoption, for instance the systematic literature review conducted by Causevic et al. [20] identified seven factors limiting industry adoption of TDD. In addition, several experiments have been conducted to ascertain the effects of TDD on internal and external quality [134], productivity [100], and the importance of programming and testing skills [48].

1.3 Research Questions and Contributions

This section outlines the goal and overarching research questions that guided this thesis and the corresponding research contributions. Detailed research questions are stated and discussed in later chapters.

The goal explored in this thesis is to increase the understanding of the phenomenon called "Software Craftsmanship" using empirical software engineering methods.

1.3.1 Research Questions

This thesis explores four overarching research questions:

RQ1 How does intensive test automation support software craftsmanship?

Although already Dijkstra stated: "Program testing can be used very effectively to show the presence of bugs, but never to show their absence" [37], much of software engineering centers around effective ways of performing software testing. Extreme Programming [7] and other Agile processes advocate Test-Driven Development [8] and state that requirements, in the form of executable test cases, should be written directly by the requirement owner.

This research question is illustrated by highlighting that the focus on developing functional tests will cause the test base to grow faster than the production code (Chapter 2). Although this carries a cost in the form of required maintenance (exemplified in Chapter 4), it also enables aggressive refactoring of the production code, as discussed in Chapter 3.

RQ2 What software craftsmanship principles and practices can be identified in the literature and a real-life, multi-year project encompassing several teams?

Given that much of the software craftsmanship practices come from books and non-peer-reviewed literature, we performed a systematic literature study to explore how the area has been addressed in research studies, and we used our case study to illustrate which practices could be seen empirically. Although this has the limitation of being a single case, in a particular setting, we adopted the pragmatic approach of "truth is what works in a given situation."

This research question is answered by the literature review and case study results in Chapter 3, summarized in the concept map, Figure 3.3, with associated detailed results.

RQ3 What effects can be seen by postponing the craftsmanship principle of cleaning up technical debt related to deprecated code usage?

This research question is the topic of Chapter 4, which describes how postponing "keeping code clean and up-to-date" in some cases caused additional debt to spread in the test base, but in other cases, the effects were benign. The main message from this chapter is that a visualization system of the prevalence and spread of technical debt would have been beneficial for judging which rules to apply.

RQ4 What are the practical implications of software craftsmanship on software development practices?

This research question is discussed in Chapter 5, where we bring up the practical implications of software craftsmanship to a broader audience.

1.3.2 Research Contributions

This licentiate thesis contributes to the software engineering field by highlighting the principles and practices of software craftsmanship. Contributions are listed per chapter.

C1 Highlighting the importance of testing in layers and managing the growth of test cases *(RQ1)*

As is illustrated in Chapter 2 and Chapter 3.5, when testing is automated, the amount of test code can be expected to grow at least as fast (typically

faster) than the production code. This means that standard software engineering approaches also apply to test code, which has to be managed as any other artifact. The chapter also introduces, but does not elaborate further on, how to manage test code refactoring by seeding production code with errors that should be found both with the original and refactored test code.

C2 Providing the anatomy of software craftsmanship, with its principles and practices, based on literature and a large-scale industrial case study. ***(RQ2)***

Chapter 3 describes both the literature search and empirical findings from a case study and uses these data sources to derive a concept map of Software Craftsmanship illustrated in Figure 3.3.

While we do not claim the generalizability of the map, we have made sure to include aspects from both literature and the case study result in the final map while keeping a traceable path to the concepts.

C3 Highlighting the importance of proper technical debt accounting and visualization techniques, exemplified via deprecated code usage in test code. ***(RQ3)***

In Chapter 4, we add evidence to the claim by Kruchten et al. [72], stating that: "The most likely cause [of Technical Debt] is schedule pressure." Furthermore, we show the importance of visualizing and accounting for the technical debt related to deprecated code usage in the different test codebases in the studied case. With IDE support for visualizing deprecated code in a strike-through font, the Java codebase showed no growth in deprecated version usage. In contrast, debt in parts of the XML code base, lacking such visualization, grew considerably for a limited time. We also show that the growth in technical debt was highly right-skewed (concentrated to a few, highly used services). This implies that rank lists, rather than measures of central tendency, should be used to manage technical debt related to deprecated code usage.

C4 Providing practical, actionable advice on how to enable software craftsmanship in an organization. ***(RQ4)***

Chapter 5 highlights the central themes of accountability, feedback loops, skills, and fostering a shared professional culture to build well-crafted software. We explicitly try to change the stereotype voiced by Boehm [13] as a

software craftsperson being a "lone cowboy programmer" by highlighting the community aspect of modern software development.

1.4 Methodology

1.4.1 Research Methods

Easterbrook et al. [39] provide guidance on what empirical methods to use when conducting Software Engineering research and what empirical answers different philosophical stances will accept as empirical truth.

The *positivist* approach, which has been the standard philosophical view of the natural sciences, states that all true knowledge must be based on logical inference from a set of basic observable facts. Typically, this requires studies that break down complex scenarios into smaller, simpler components.

The *constructivist* (or *interpretivist*) approach [70], in contrast, states that scientific knowledge has to be viewed from its human context. It emphasizes the fact that social theory terms are socially constructed. Hence, interpretations of what a theory means (in a particular context) are equally important as the empirical observations on which it is based.

Adopting *critical theory* [19], on the other hand, would view research as a political act, as knowledge acquisition empowers different societal groups or entrenches pre-existing power structures. Therefore, critical theorists typically engage in participatory research methods, striving to influence or steer the direction of the studied subjects.

In contrast, *pragmatism* [94] takes an engineering approach to science, acknowledging that knowledge is approximate, judging its value by its practical problem-solving utility: "truth is whatever works at the time." Many pragmatists acknowledge the need for consensus; truth can be found during rational discourse, while the participants weigh whoever has the better arguments. Pragmatism also explicitly acknowledges that usefulness is relative; what is useful to one person (or organization) might not be useful to another.

This thesis focuses mainly on descriptive, exploratory studies and is most closely aligned with the pragmatic approach. Through our case study we do not aim for statistical generalizability but to achieve analytical generalizability, via cross-referencing the literature results, and by validating the consolidated results with the studied subjects. Likewise, we have not delved into action research inspired by critical theory, as the events are, in effect, described *post factum*, without any particular research influence. While the author was part of

the studied organization, he was not a researcher and was not aware of action research philosophy or methods.

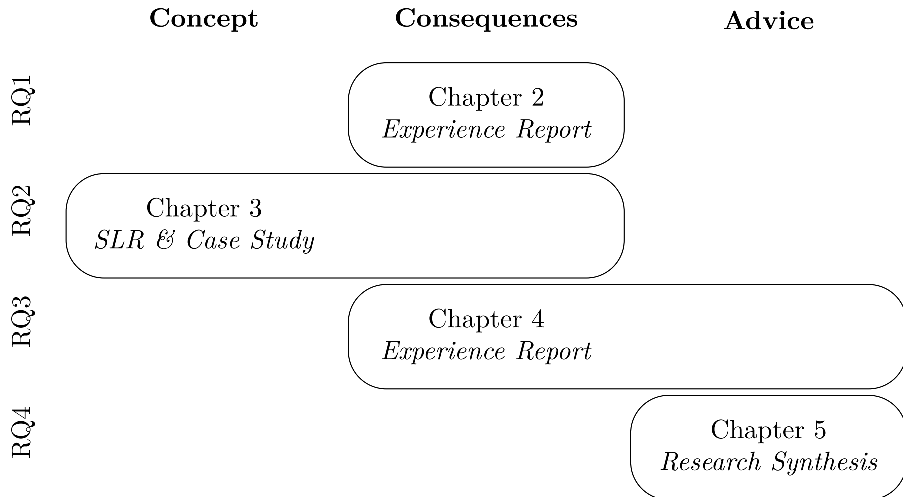


Figure 1.1: Research Methodology Overview.

As illustrated in figure 1.1, the studies described in this thesis were conducted using both a fixed design, in the form of a Systematic Literature Review, and a flexible design, in the form of a Case Study, using mixed-methods research tools. The experience reports mainly used quantitative data, processed using version-controlled scripts and standard data processing tools.

Literature Review

In 2004, Kitchenham et al. [69] proposed adopting an evidence-based approach, much like the medical sciences, rather than relying on expert opinions or anecdotal evidence. Performing a Systematic Literature Review (SLR) [68] is one proposed method to gather available evidence for a particular research question. Other methods, such as Systematic Mapping Studies [104], can also be used and have various pros and cons relative to an SLR.

We used Wohlin's snowball approach to SLRs [140], with forward (citations) and backward (references) snowballing phases. Based on a search string in

Google Scholar, we selected nine starting papers (the start set). According to Wohlin, the start set should: (i) include papers from all relevant communities (clusters), (ii) not be too small, (iii) if too large, can be reduced by including only the most relevant and highly cited papers, (iv) cover diverse publishers, years and authors, and (v) be formulated from keywords in the research question. After performing two rounds of seed selection and iterating four snowball phases, we found only nine additional research papers, of which few were peer-reviewed. We augmented our search with the selection of books that were referenced by the found papers. The full results are found in Chapter 3.4.

Case Study

We used the case study methodology described by Runeson et al. [114] in an exploratory and descriptive setting, with a single unit of analysis (the studied project, which we followed for seven years). Alternative approaches could have been action research [78], ethnography [122], or grounded theory study [53]. Following Robson and McCartan [111], choosing action research would have been suitable if we were conducting research in order to change, improve or influence the observed unit; ethnography would have been appropriate if we had planned the study before being immersed in the studied case; and grounded theory would have been appropriate if we were to develop a new theory of a particular research question or smaller area, such as "Why developers care about or ignore the continuous integration results?" In contrast, the field of Software Craftsmanship (as illustrated in Chapter 3) is vast, encompassing a wide breadth of software development practices. Therefore, we adopted the case study approach, which explicitly states that multiple sources of evidence are used [142]:

> A case study is an empirical inquiry that investigates a *contemporary phenomenon* (the "case") *in depth and within its real-world context*, especially when the *boundaries between phenomenon and context* may not be clearly evident.
>
> A case study inquiry copes with the technically distinctive situation that there will be *many more variables of interest than data points*, and as one result relies on *multiple sources of evidence*, with data needing to *converge in a triangulating fashion*, and as another result benefits from the *prior development of theoretical propositions* to guide data collection and analysis.

We mined source code and test repositories, defect tracking systems, and wikis to obtain quantitative evidence. To increase validity, we used data triangulation (several methods of data collection), observer triangulation (several researchers participating), and methodological triangulation (using both quantitative and qualitative methods). Robson and McCartan [111] notes that while triangulation can increase validity, it can also raise logical and practical difficulties—for instance, if the data from different sources are contradictory or hard to compare directly. While analyzing data, we highlighted contradictory or non-confirmatory evidence, reporting this in our findings.

We used thematic coding [14] to analyze qualitative data, as exemplified by Cruzes and Dybå [29]. We used the *Integrated Approach*, i.e., we started with a deductive set of codes (based on reading literature and case study experience) and allowed the creation of new codes as the responses (interview data and literature) were processed.

Experience Reports

Compared to the case study and systematic literature review studies, the experience reports in this thesis followed a less stringent method. In order to increase the correctness of our findings, we kept an audit trail by keeping the analysis and collected data in a version-controlled repository. We used standard statistical processing tools (scripts, the R programming language, and RStudio) to process data and generate figures.

We increased the validity of the conclusions by presenting the findings to the studied organization and receiving feedback, albeit in an unstructured format. In some cases, this led us to rephrase and clarify the presentation of our findings.

Research Synthesis

In Chapter 5, we synthesized the findings, mainly from Chapter 3, adding some additional conclusions to give more practitioner-oriented advice. Cruzes et al. [30], [31] highlight the importance of case studies synthesis, and we hope to be able to perform cross-case analysis in future work.

1.4.2 Limitations and Threats to Validity

The validity of research is central to the trustworthiness of the results and the generalizability of the findings [114]. In Software Engineering, validity is

commonly discussed from four angles: *construct validity*, *internal validity*, *external validity* and *reliability*. However, other perspectives are also possible. Maxwell [90] advocate a typology for flexible design studies such as case studies that focuses on the three types *description*, *interpretation* and *theory*. In this thesis, each study (chapter) reports on the detailed threats to validity. For the overarching description, we will adopt the classification commonly used in Software Engineering [141].

Construct validity deals with the extent to which the operational measures really represent what the researcher had in mind (i.e., what the research questions state). Questions may be interpreted differently; a measurement device such as a thermometer is a bad predictor if the research question concerns whether or not to bring an umbrella for the lunch walk, and so on.

Throughout this thesis, we used various quantitative measures elicited from source code repositories, defect tracking systems, and requirements handling systems. When dealing with such measures, it is important to assess both the correctness (including completeness) of the data within the system and how the derived measures reflect the research question.

Chapter 2 presents quantitative measures related to code size (volume) and defects (internal and external), together with author experiences. The main threat to construct validity is whether the quantitative measures, such as test base size, can be used to assess the amount of craftsmanship or if there were other consequences that caused the large growth in test code. We increased the construct validity of the measures by presenting the findings for parts of the studied organization and getting feedback.

Chapter 3 includes more qualitative data, including interview guides, transcripts, and literature. We increased the construct validity by letting two additional researchers play “devil’s advocate,” taking a critical view of the proposal of the first author. We also presented intermediate versions of the concept map to parts of the studied organization.

We took a similar approach in chapter 4, primarily built on quantitative data from the source code repository. The main threat to the construct validity here is whether the quantitive metrics (e.g., presence or absence of a deprecated item in a file) are related to the concept of technical debt and how this is related to craftsmanship. Five developers with experience from throughout the studied period were interviewed to assess the conclusions and the validity of the measures.

Internal validity deals mainly with causal relation (A causes B, or does not cause B). When investigating causal relationships, there is always a risk that

unknown confounding factors influence outcomes, and internal validity concerns how to minimize this risk.

In the studies in this thesis, we mainly deal with descriptive and exploratory findings and make few causal inferences. We validated the conclusions regarding the large test base and its connection to the fact that the organization used their self-stated "Test-Focused Development" principle by presenting the findings to a subset of the studied organization.

External validity (or generalizability) deals with to what extent it is possible to generalize the findings and whether the findings are interesting for other people outside the studied case. All of the studies in this thesis were performed in a particular project in a single company. At face value, this would limit the external validity. However, we have also incorporated other viewpoints by conducting a multi-vocal systematic literature review, bringing in perspectives from other organizations. Nevertheless, the application of our findings might be different in other situations, such as where requirements are more fixed (either due to legal regulations, the laws of physics, or other rigid constraints).

Reliability is concerned with whether and how the data and analysis depend on the specific researchers. As stated in the individual chapters, the author of this thesis was engaged in the studied organization for a large part of the studied period, which is a significant threat to validity. To counter this threat, we used two impartial researchers during data gathering (e.g., interviews) and analysis. We also kept an audit trail (i.e., a record of activities) while conducting the studies. While parts of this trail are public (e.g., the SLR findings and interview questions, see Chapter 3), other parts (e.g., interview transcripts, coding system) are available to reviewers upon request.

1.5 Conclusion and Future Research

We note the general overlap between Software Craftsmanship, Agile and Lean, as framed in the leading books on the subjects. However, there are also overlaps with modern Software Engineering, in particular Behavioural Software Engineering [77], where researchers strive to describe and understand the thoughts and decision processes of software developers, teams, and organizations from a psychological perspective.

However, there are also differences. The code kata concept, and the deliberate practice it espouses, have only a few research articles listed on Google Scholar and seems to warrant a deeper, more systematic study. The fact that the lead developers in the studied case were cognizant of the importance of

the shared professional culture—and therefore demanded the initial outsourced teams to be physically located at the main site while being imbued in the professional culture—should also be studied more systematically, as this goes against the general outsourcing state-of-practice.

Is Software Craftsmanship a mere Platonic ideal, unobtainable for those of us stuck down in the day-to-day, coding-for-a-living cave? Perhaps—but this does not prevent us from studying the shadows on the cave wall, trying to draw conclusions, and, not the least important, trying to instill a sense of professionalism in the next generation of programmers, wherever they are physically located.

We do believe that our anatomy map also incorporates the need for continuous improvement. Indeed, we are confident that future versions of the craftsmanship principles and practices will have to consider how to structure feedback, verification, and validation loops when dealing with scarce resources such as long-running Machine Learning or Artificial Intelligence models or mastering the qubits in a physically manifested quantum computer. Given the increasing complexity and importance of software in the world, we are certain that the need for professionalism will grow, particularly where physical objects are affected.

Chapter 2

Test-Driving FinTech Product Development

This chapter is based on the following paper:

A. Sundelin, J. Gonzalez-Huerta, and K. Wnuk, "Test-Driving FinTech Product Development: An Experience Report," in *International Conference on Product-Focused Software Process Improvement*, Springer, 2018, pp. 219–226, ISBN: 978-3030036737. DOI: 10.1007/978-3-030-03673-7_16

Abstract

In this paper, we present experiences from eight years of developing a financial transaction engine, using what can be described as an integration-test-centric software development process. We discuss the product and the relation between three different categories of its software and how the relative weight of these artifacts has varied over the years. In addition to the presentation, some challenges and future research directions are discussed.

2.1 Introduction

Software and software products are the critical elements of the Financial Technology (FinTech) [76] revolution that reshape the way how individuals and financial institutions save, borrow, make payments, and manage risk [49]. Software

is enabling societal change in our relationship with money, especially in developing economies where alternative financial services are more customer focused and allow more people to have access to finance without the need of a bank [12]. Ericsson has seen the opportunity that FinTech offers as early as 2010 and decided to create a financial product for developing economies that provides access to payment services to users without credit card or bank account. Ericsson developed the product considering market demands and requirements such as *security*, *auditing*, *correctness*, *performance*, *availability*, *flexibility*, *fast time to market* and *development efficiency.*

In this paper, we discuss experiences on how Ericsson tackled the problem of crafting a software product for the financial sector not only migrating to a modern programming language and with a Service Oriented Architecture, but also using modern ways of developing the software such as *test-driven development*, *integration tests*, *continuous integration*, *clean code*, *learning by doing*, *mandatory solution review* and *simple communication.* Several studies analyze the effects that TDD has on code quality and defect rate [89], [98], though few studies analyze the long-term effects that TDD might have in the project, regarding the number of defects and the size of the test base as compared to the code base. Moreover, there is a lack of longitudinal experience reports of developing Fintech products for global markets.

2.2 Background and Related Work

Although it has earlier roots in the Smalltalk community, the term Test-Driven Development was popularized in the late 1990s and described as part of the Extreme Programming process, [7], [8]. Several scientific studies have analyzed the effects of TDD e.g., [40], [89], [98].

In an experiment described in [47], the authors compare TDD with the alternative iterative test-last (ITLD) process, concluding that the claimed benefits of TDD arise not from its test-first approach, but from the fine-grained, steady steps, with fast feedback that improve focus and flow. Our paper supports this conclusion, adding aspects of product development over eight years.

One of the first public tools to support automated acceptance tests was the Framework for Integrated Tests (Fit) [1]. The acceptance-test-driven development (ATDD) process [107] was studied in [57], [93].

Most of the studies of TDD have focused on shorter timescales, from some weeks, up to a few years worth of software development. This paper adds

[1] http://fit.c2.com/

experiences from eight years of building a product from scratch using rigorous testing methodologies, and the effects that this has had on the code base.

2.3 Case Description and Analysis Method

The system under study forms part of a FinTech global product that enables access to financial services via mobile phones and the Internet. It is typically installed in a high-availability configuration, with geographical redundancy, to meet service uptime requirements. The system is a transaction-intensive application, with incoming and outgoing interfaces, a database, and scheduled tasks such as the sending of notifications. As it is a financial application, security has played a central role in its development.

The studied system consists of the financial core of the application, containing the core business logic, such as the financial transaction management. The core exposes its services via a set of *requests*, similar in spirit to the system calls of the Unix kernel. There are other components in the product, such as user interfaces, both graphical and textual, but these are not studied in this report. All other components use the services of the core in order to perform their tasks. The system is built in Java, using EJB 3[2] principles and is deployed using a custom, light-weight EJB container, also exempt from this study.

One of the guiding principles when developing this application was intensive test automation. Testing should take place in different layers, with the bulk of the tests in the lower layers (unit tests), and progressively fewer tests the higher the abstraction level. This follows the principles outlined in the testing pyramid [25].

2.3.1 Studied artifacts

We analyzed the production code and test artifacts developed by the *development team* during feature development. Ericsson also has dedicated *testing teams*, focusing on testing the complete system, including all required special hardware, such as hardware cryptography modules and application firewalls, but these activities are not studied in this report.

The studied software is classified according to the following three categories:

- *Production code* - which is deployed at the customer site, and perform some useful action in live deployed systems.

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

- *Unit test code* - which is developed alongside the production code, typically by the same developer.
- *Integration test code* - using the externally visible interfaces of the system, typically describing the use cases that the application shall provide. Integration tests are developed by the same team and at the same time as the production code, though typically by different developers.

In addition to these three categories, the system also contains small amounts of other software and configurations files, such as installation support software, test enablers that aid integration testing, and system test code for testing the entire software system.

The number of authors of the studied software has varied over time, due to business and organizational changes. For the production and unit test code, the number of authors has been between 9 and 74, with a median of 35. The integration tests have a slightly wider span, between 8 and 79 authors, also with a median of 35. The majority of developers (median: 29.5) has developed both production code and integration tests.

2.3.2 Tools

We use the common open-source tools cloc[3] and Git[4] in order to calculate statistics for the above-mentioned categories. We collected statistics for every feature-enhancing release made of the product, plus three initial "pre-releases", denoted P01, P02, P03, before the first commercial release, denoted R01, in mid-2012. On average, 81 days have passed between releases, with a median of 62.5 days and an IQR of 55.5 days. There have also been other releases made of the software, both for error corrections, and candidates meant for internal testing.

2.4 Results and Discussion

2.4.1 Size of the code base

Table 2.1 lists the mean and median lines per file and Figure 2.1 illustrates the number of lines of code per category, for each studied release. The last studied release, R32, consists of about 5000 files of production code, and about half as

[3]https://github.com/AlDanial/cloc

[4]https://git-scm.com/

Table 2.1: Mean and median number of lines per file.

File type	Mean lines/file	Median lines/file
Production code (Java)	90	55
Unit test code (Java)	228	163
Integration test (XML)	133	97

many unit test files. The number of integration test files is more than double the number of production files, about 12000.

Figure 2.1 shows a linear growth, with integration test code (red) dominating over production code (green). While the production code has grown from about 42 kLOC in 583 files in the first preliminary release, to about 460 kLOC in 5166 files in the last studied release, the number of lines of integration tests has grown to 1488 kLOC, in 11211 files. The first two preliminary releases contained no integration tests but were tested using other tools, later discarded due to lack of productivity. To enable efficient integration testing, the developers chose to develop an XML-based testing tool. The tool was first used in the P03 release, which comprised of 110 kLOC integration test code lines, distributed in 622 files.

Regarding the unit test code (blue), we see that starting from release R11, the number of lines of unit test code exceeds the number of lines of production code. In the latest studied release, there are about 30% more lines of unit test code than of production code. Thus, the unit test code base is also growing faster than the production code, though not as fast as the integration tests. The fact that the typical unit test file is larger than the typical production code file supports the notion that the tests are mostly concrete code, in the typical *Arrange*, *Act*, *Assert* fashion, whereas the production code consists of a higher number of interfaces and abstract classes.

We can conclude that this is a product where test code, both unit tests, and integration tests, make up the bulk of the software. As the growth rate of both the unit tests and the integration tests are higher than for production code, it becomes a necessity to manage this growth, for instance by reducing duplication and increasing modularity and reusability. The importance of this increases as the product ages and grows.

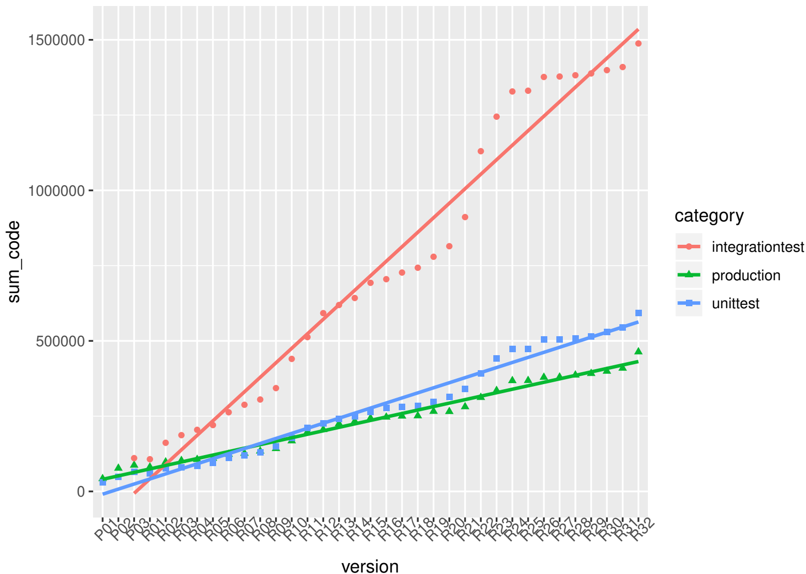


Figure 2.1: Lines of code for each category, per release, P01, P02 and P03 are initial prereleases and R01 is the first commercial release.

2.4.2 Defect prevalence

Figure 2.2 shows the number of corrected defects per release. The upper (red) line is the total number of corrected defects, the middle (green) line is the number of defect corrections that are new to the release, and the lower (blue) line is the ratio between the defects new to the release and the size in kLOC of the production code base.

The defects in this statistic include those found by customers in the field, internal testing organizations during system verification, and those found by the developers after the release of a feature. Defects found by developers during the development of a feature are not included. No goals or penalties related to the number of found defects in the product have been used by the organization, though goals related to the defect response time have been used. Thus, it is unlikely that developers have refrained from issuing defect reports in order to fulfill some target objective.

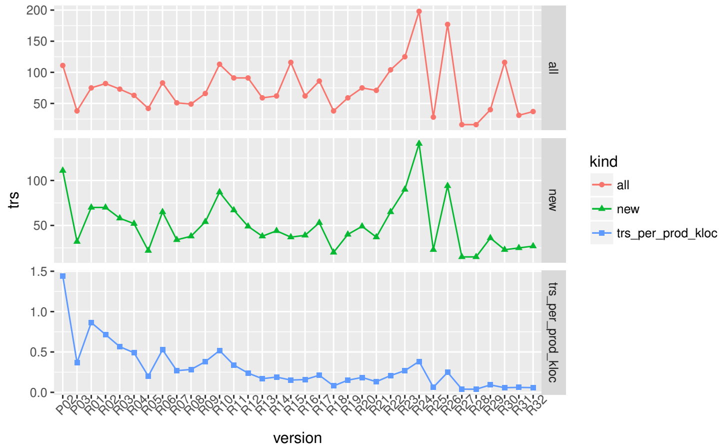


Figure 2.2: Number and ratio of corrected defects for different versions.

It is quite evident from the picture that the ratio of defects was higher at the beginning of the product lifecycle when there were few lines of production code, and none or few integration tests. It is also quite evident that some releases contain more corrections than others. There are two reasons for this: Some releases (e.g. R24 and R26) contained many corrections from customer branches, which was then integrated into the main branch. Some releases (e.g. R25) was made very close in time to the prior release, causing it to contain fewer changes. Further analysis of the defect origin is deferred to another paper.

2.4.3 Changes to the code base

By using *git diff* to analyze files changed in each version, it is possible to get an overview of the changes in the code base. It should be noted that this is only the “net change”, as this statistic does not capture files that have changed multiple times between versions. Also, lines are reported regardless of whether it is a code or comment line, and a changed line is counted as both an added line and a removed line.

On average, between versions, production code have added 28372 lines and removed/changed 14535 lines in 884 files, unit tests have added 32891 lines and removed/changed 12934 lines in 565 files, and integration tests have added 75005 lines and removed/changed 39424 in 2530 files. This is another way of

illustrating that changes to the integration tests dominate over the production and unit test code.

2.4.4 Guiding Principle: Test fast, test in layers

While the ATDD and TDD processes were encouraged, explained and exercised during onboarding of new developers, it was still up to each developer to do their tasks in the order they preferred. Thus, it is likely that some developers followed other processes, such as Incremental Test-Last (ITL) [47]. The common ground between these two processes is that tests should be developed as close (organizational and temporal) to the production code as possible, and refactorings should be performed when all tests succeed. This is in contrast to a more traditional "Design-Implement-Test" approach, where typically the tests are developed once all, or most functionality is implemented.

The organization actively required that tests were developed as the requirements were implemented. No feature was allowed into the product without having the required supporting test base. Also, there were continuous discussions among developers, how a feature should best be verified, and what things that were the most important to verify. As is shown in the defect statistics, figure 2.2, the integration test base helped limit the number of defects as the product has grown. One of the consequences of this principle is that the amount of test code will grow, in parallel with the growth of the production code. We see in the reported statistics that both the unit tests and the integration tests grow faster than the production code.

Both the number of files and the number of lines of integration tests is much greater than the corresponding metrics for production and unit testing code. In part, this stems from the use of XML as a specification language for the integration tests.

One obvious benefit of having a separate language for integration tests is that it is possible to enforce certain rules, by only implementing wanted features in the language executor. For instance, in the current integration test language, there is no support for conditional branches and only limited support for iterative loops. Another benefit is that developers specifying the integration tests have a clear line between the production code/unit tests and the integration tests. They can safely ignore the Java syntax and features in the Java language.

The most severe disadvantage of the separate integration test language is that the lack of effective module support causes the number of lines of integration tests to grow faster than the production code and unit test code. Another disadvantage is that there is no IDE support, such as code completion, refac-

toring or debugging support, out of the box. Thus, trivial transformations or reports such as *rename method*, or *find usages* becomes difficult for developers not well versed in file system and text processing tools such as *find*, *grep*, *awk* or *perl*. Due to the lack of code completion, there is also the risk of longer development times, and developers being unaware of similar functions for setting up the scenarios.

2.5 Implications for Research and practice

As can be devised from the statistics shown in Section 2.4, the product currently consists of considerably more lines of test code than production code. In order to be able to work efficiently and develop with speed, it is imperative that this test code is kept clean and undergoes a similar refactoring scheme as the production code. A design principle used when developing test cases is that each test should be "self-contained", and "self-specified". Each test case should specify its required state, and asserts should reference this state, and not obscure references to other "magic numbers". The disadvantage of "the self-containment principle" is that naïve developers may copy code, instead of extracting sections into methods or modules.

Care has to be taken when refactoring test code. In particular, it is important that the principles of *Arrange*, *Act*, *Assert* continues to hold for each test. Also, each test case should continue asserting everything it asserted before the refactoring, to avoid causing the refactored test case to be more lenient than the original one. The *Arrange* phase, however, should be as lenient as possible, only specifying the minimal state required to make the test succeed. Different initial states are typically referred to as fixtures, and to avoid undue repetition, it is important to keep track of which fixtures that are already available when writing new test cases.

A solution to the test refactoring problem is to introduce known errors in the production code while refactoring the test code, checking that both the old and new test cases catch the introduced errors. Once a satisfactory refactoring has been completed, the new, refactored test case is committed, and the introduced errors reverted, leaving the original (non-faulty) production code. This field should be studied more thoroughly, e.g., whether the refactoring validation process can be automated, or if some static rules could be devised.

Chapter 3

Towards an Anatomy of Software Craftsmanship

This chapter is based on the following paper:

A. Sundelin, J. Gonzalez-Huerta, K. Wnuk, *et al.*, "Towards an anatomy of software craftsmanship," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, to appear, 2021. DOI: 10.1145/3468504

Abstract

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 article 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.

3.1 Introduction

The notion that programmers should be responsible for what they produce has early roots. Already in 1975, Brooks [17] 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 [91], formalizing the software craftsmanship concept, and since then, several books have been written on the subject [83], [86]–[88]. Another early inspirational work was published in 1999 by Hunt and Thomas [61].

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 article 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 3.4.

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

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 article is structured as follows: In Section 3.2, we give the background and related work of software craftsmanship and define the terms we use throughout the article. In Section 3.3, we report on our research methodology, with Section 3.3.1 focusing on the systematic literature study, Section 3.3.2 focusing on the case study methodology, including the studied context, and Section 3.3.3 focusing on the process of building the anatomy. In Section 3.4, we report on the results of the **Systematic Literature Review (SLR)**, and in Section 3.5, 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 3.6, we discuss the implications for the software development community at large. In Section 3.7, we discuss the threats to the validity of the study. In Section 3.8, we draw on the analysis, outline future work and research directions, and make conclusions.

3.2 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 [17], Hunt & Thomas [61], McBreen [91], Martin [86]–[88] and later also Mancuso [83]) 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 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 SLRs (e.g., [64], [115], [135]), Systematic Mapping Studies (e.g., [35]) and even Tertiary Studies (e.g., [59]) 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., [65], [38], and [1]). Likewise, multiple studies address the potential benefits and drawbacks of **Test-Driven Development (TDD)**, with several experiments (e.g., [47], [134]), case studies (e.g., [38]), and SLRs (e.g., [97])

Lean Software Development was popularized by Poppendieck [106] and has been studied in an industrial setting [103], [105]. Several SLRs and Systematic Mapping Studies report results on metrics related to Agile and Lean software development and their relevance in the software industry [21], [41], [74].

3.3 Research Methodology

This article uses the SLR method, using Wohlin's snowballing approach [140], and a case study method following guidelines by Runeson et al. [114]. 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 an SLR. 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.

3.3.1 Systematic Literature Review Methodology and Execution

We conducted an SLR using the snowballing method described by Wohlin [140]. We used a hybrid search strategy by combining the database search with iterative citations and references analysis [96]. Forward snowballing (citation analysis) greatly improves the precision, while backward snowballing (references analysis) greatly improves the recall of literature reviews.

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 papers only referring to other works, such as [87], 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 [140], 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.

Snowballing iterations

We performed four snowballing iterations summarized in Table 3.1 and stopped when we found no new relevant papers, applying the inclusion and exclusion criteria following the process described in Section 3.3.1. 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 [51]. 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 3.4 contains the full results of the SLR.

3.3.2 Case Study Methodology

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

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.

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

Table 3.1: Snowballing Iteration Statistics and Results

Iteration	Number of citations and references screened	Included papers and books
Seed-1		P1 [132], P2 [99], P3 [112], P4 [81]
Seed-2		P5 [82], P6 [63], P7 [27], P8 [108], P9 [84]
Iteration 1	213 references and 186 citations	P10 [102]
Iteration 2	30 references and 1 citation	P11 [79], P12 [133], P13 [101], P14 [13]
Iteration 3	217 references and 517 citations	P15 [119], P16 [10], P17 [80]
Iteration 4	18 references and 78 citations	P18 [139]
Ref. Books	146 referenced books	B1 [17], B2 [91], B3 [75], B4 [87] B5 [121], B6 [60], B7 [120], B8 [88] B9 [83], B10 [56], B11 [138]

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.

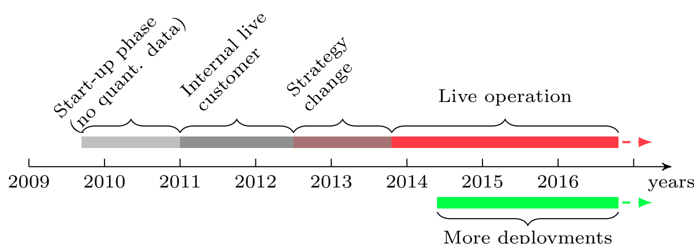


Figure 3.1: Timeline of major events in the studied system.

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

Figure 3.1 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 3.5.

During the studied period, 155 individual developers have contributed to the studied system (measured via the Git *Author* tag). The first author of this article was a developer from the project start until October 2016. Table 3.2 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 3.2: Quarterly Developer Statistics

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 [22]. 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 article.

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 3.3 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.

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

Table 3.3: 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

3.3.3 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 and Clarke [14]. 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 [53], that in general is better suited to answer broader questions, such as *"what is going on there?"* [127]. TA is a robust and systematic framework for coding and analyzing qualitative data, identifying patterns across datasets in relation to research questions [15]. 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[14] by starting with a *theory* (a set of codes and themes), updating this as new data emerged.

Figure 3.2 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 3.5. 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 gray 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 [116] 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 3.2. This process triggered the need to review the already coded materials to identify potential instances of the new codes and themes in the data.

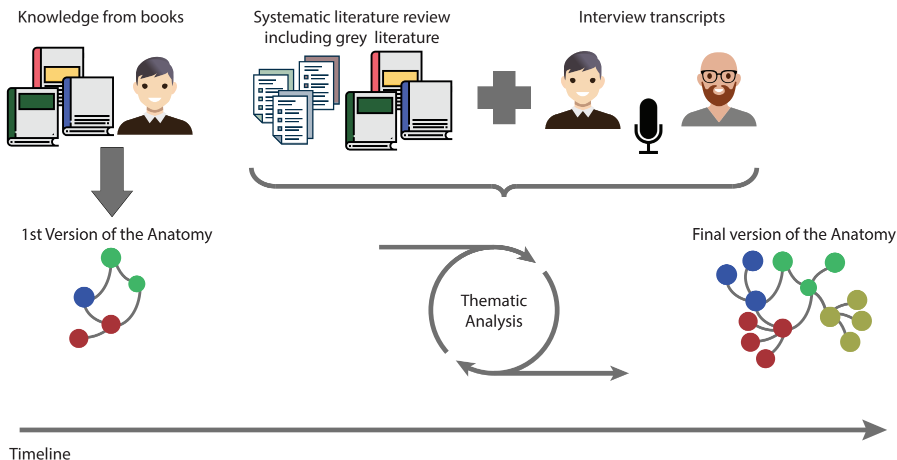


Figure 3.2: Process for Building the Anatomy of Software Craftsmanship.

Taking the “Requirements” concept as an example to illustrate the process:

1. The first author of this article 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 3.5. Based on this, he initially decided on the code *On-site customer*, as it is a concept from XP [7] 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 [91] and B8 [88] 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 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 3.11 for details. In B2 [91], 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." [33]

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 [91] and papers P3 [112] and P9 [84].

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 article 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.

3.4 Systematic Literature Review Results

In this section, we summarize the main findings of the SLR. Table 3.4 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 [62]. 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

Table 3.4: Papers Resulting from the Systematic Literature Review

Paper [ref]	Found in	Refs	Cited	Rigor	Relevance	Venue	Year	Empirical	Main contribution
P1 [132]	Seed1		P10	0	0	Journal	2003	No: vision paper	Craft metaphor for software creation
P2 [99]	Seed1			4	3	Journal	2013	Yes: qualitative and quantitative, longitudinal study	Craftsmanship forums and chats as a part of community of practice
P3 [112]	Seed1			3	3	Conf.	2013	Yes: questionnaire and focus groups	Community of practice as a part of software craftsmanship
P4 [81]	Seed1			2	3	Conf.	2014	Yes: qualitative interviews and focus groups	Different conceptualizations of craft in building software
P5 [82]	Seed2			1	4	Workshop	2016	Yes: experience report	Analyzes software craftsmanship values in a Scrum project
P6 [63]	Seed2			0	2	Magazine	2014	No: opinion paper and anecdotal evidence	Engineering is craft supported by a theory
P7 [27]	Seed2			0	2	Non-academic conference	1994	No: experience report mostly based on anecdotal evidence	Stresses the importance of craftsmanship
P8 [108]	Seed2			0	2	Non-academic journal	2003	No: opinion paper	Discusses general craftsmanship and software craftsmanship models
P9 [84]	Seed2			1	2	Workshop	2008	No: personal experience	Focus more on agile than craftsmanship
P10 [102]	Iter1	P11, P12, P13, P14	P1	2	3	Journal	2015	Yes: qualitative and quantitative surveys	Community of practice and software design
P11 [79]	Iter2		P10, P16	1	2	Journal	2013	No: theoretical	Epistemology of craft in modern programming
P12 [133]	Iter2		P10	0	1	Non-academic journal	2010	No: opinion paper	Katas as a part of craftsmanship
P13 [101]	Iter2	P15	P10	2	2	Magazine	2014	Yes: experiment using katas	Katas as a way of learning and personal improvement
P14 [13]	Iter2		P10, P16, P17	0	1	Conf.	2006	No: opinion paper	The birth of the crafting paradigm preceding SE in the 1960s
P15 [119]	Iter3		P13	1	2	Conf.	2012	No: personal experience of the course instructor	Courses that involve craftsmanship practices
P16 [10]	Iter3	P11, P14, P17	P18	0	1	Conf.	2016	No: observations of the authors	Professional practice is craftswork
P17 [80]	Iter3	P14	P16	1	1	Workshop	2012	No: previous version of P11	Previous version of P11
P18 [139]	Iter4		P16	0	1	Journal	2008	No: personal opinion paper	Mentions craftsmanship in the history of SE

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 be an experience report from a project manager’s point of view, providing quantitative analysis of technical debt (number of lines removed over time) and velocity in backlog hours versus tool 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 3.5 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

Table 3.5: Books Resulting from the SLR. Boldface Books Influenced Initial Anatomy.

Book [ref]	Cited	Year	Main contribution
B1 [17]	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 [91]	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 [75]	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 [87]	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 [121]	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 [60]	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 [120]	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 [88]	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 [83]	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 [56]	P10	2014	Describes best practices and lessons learned while teaching the four rules of simple design [43] via code kata exercises for various groups of people over the course of five years.
B11 [138]	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[52]

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 article 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.

3.5 The Anatomy of Software Craftsmanship

In this section, we present the concept map, synthesized from the analysis of the case study findings and the SLR results. Figure 3.3 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.

3.5.1 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 3.6. To enable the value-focused architecture,

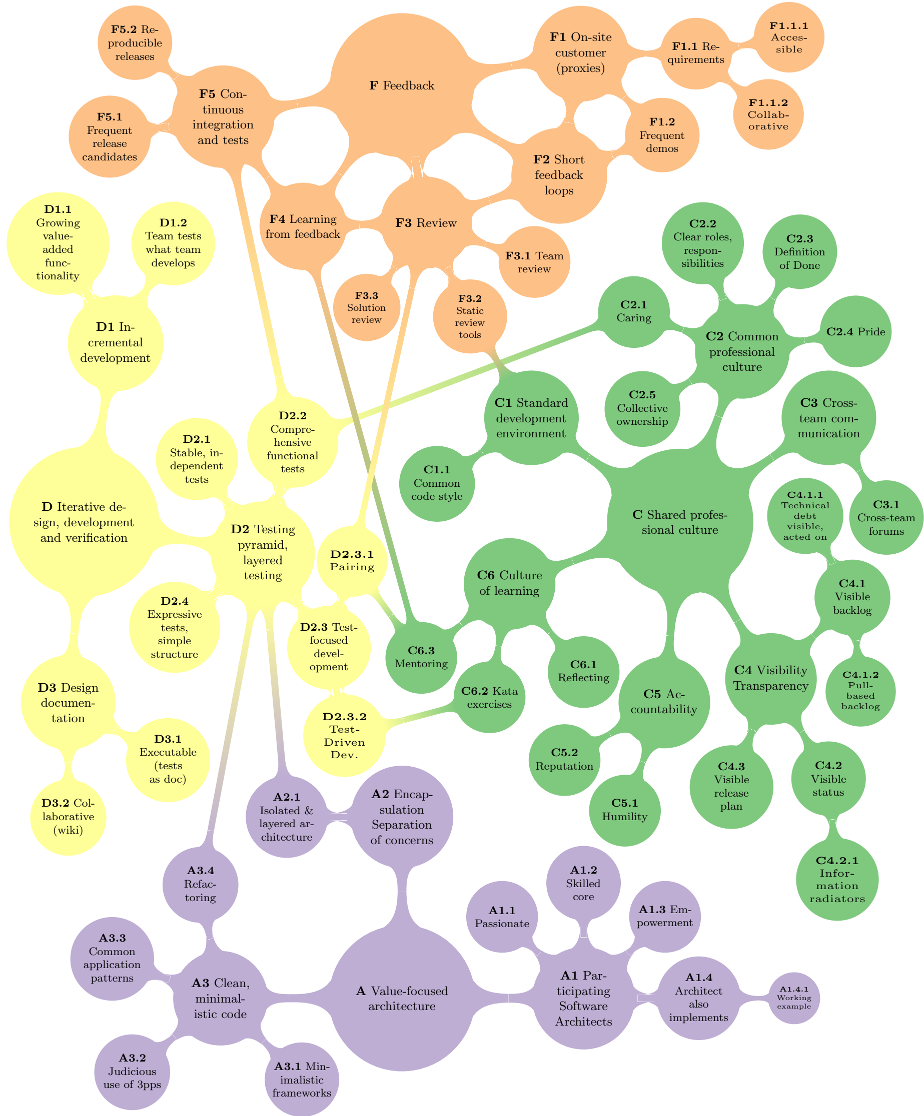


Figure 3.3: The anatomy of Software Craftsmanship.

Table 3.6: 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

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 [46] 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 [61] calls the role "technical head," tending to the big picture, and Martin [86] states that software architects need to participate in the development to spot problems and guide directions.

  Books B1, B3 and paper P3 refer to team empowerment in the context of cross-functional teams [92], 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*.

The importance of skills and passion for the craft is discussed in seven books and eight papers, as depicted in Table 3.6, 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 [16].

  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. [6] 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 principles: "the separation of concerns" [36]. While developing a complex system, there is a need to develop and evolve different parts of the system independently [6]. The layered architectural pattern is the most widely spread practice for architectural subdivision [6], [18]. 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 [110]. 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.

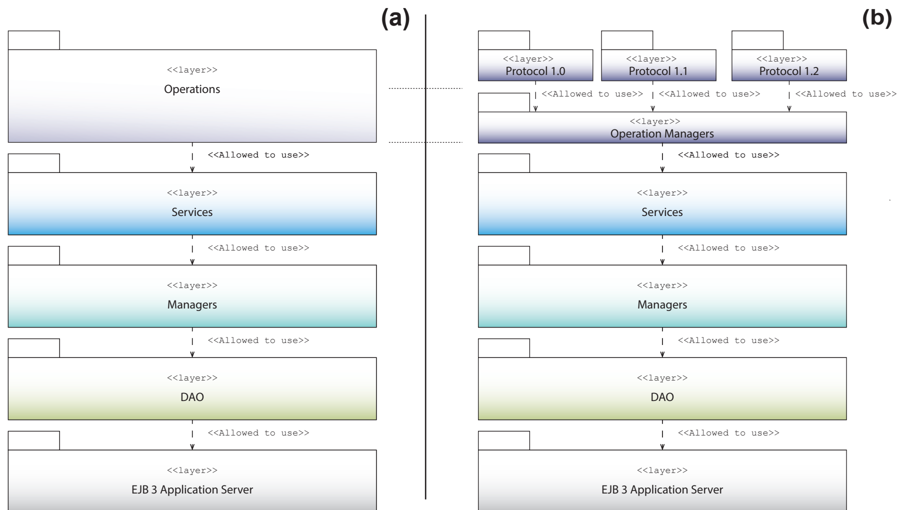


Figure 3.4: Layered view of the Initial architecture (a) and Layered view of the Architecture after separating protocols from business logic (b).

Figure 3.4 (a) depicts the initial layered architecture using UML[8] stereotypes packages as layers and stereotyped *allowed-to-use* UML dependencies, as suggested in [23]. 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 3.4 (b). The old “Operations”

[8]http://www.uml.org

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)

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 3.6, 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, [86] 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 [50].

  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 [61]. Paper P3 also mentions *refactoring* as a principle of software craftsmanship. According to Fowler et al.[46], 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[9] communities have championed the use of *kata-like exercises* to spread common idioms for developers to be productive.

[9] https://en.wikipedia.org/wiki/APL_(programming_language)

- **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[10]) 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."

  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 3.7 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

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

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 3.7: 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 3.7, 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 interquartile 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. [67], [143]).

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.

3.5.2 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 3.8 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[88] 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 3.8: 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 3.8. 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 3.9 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 3.9: Summary Code Statistic for the Five Major Code Types

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 3.8. 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 *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 [28] 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 [61] and [86] 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 [25]. While some developers preferred TDD, 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.

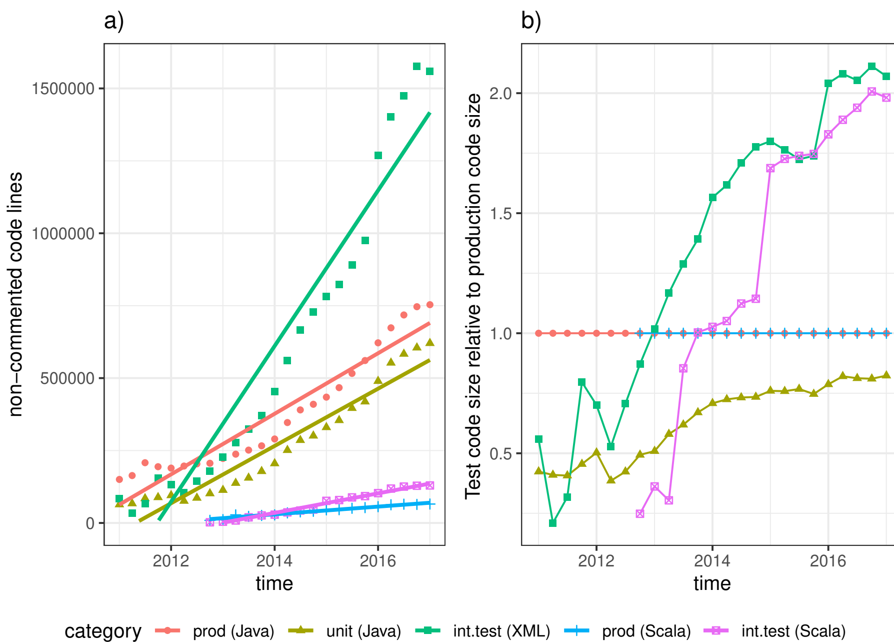


Figure 3.5: Ratio of test code vs. production code over time.

Figure 3.5a shows the numbers of non-commented source code lines for the production code (*prod (Java)*), unit tests (*unit (Java)*), inte-

gration tests (*int.test (XML)*), web GUI (*prod (Scala)*) and web GUI integration tests (*int.test (Scala)*), and Figure 3.5b 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 3.9, all five codebases grew linearly throughout the studied period. The three dips in integration test size during Q1 2011, Q1 2012, and Q4 2016 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 [95]. At the end of the studied period, 8,327 integration tests, 18,412 database-interacting unit tests, and 5,328 "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 3.9 and Figure 3.5. 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. [11] 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 when developing **application programming interfaces (API)**.

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* [71]. 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 [61] 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*[11] 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.

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

> 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.
>
> 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.

3.5.3 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 3.10, we have found evidence that a shared culture of *learning*, *caring*, *accountability* and *transparency* is beneficial and aligns with the craftsmanship approach.

Table 3.10: 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	

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[12].

  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*

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

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 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 [61] also mention the *broken window theory*, first formalized by Wilson and Kelling [66], 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**

(CoP), 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 [61] 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 DoD 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 (PO, 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 *Definition of Done* concept has been studied before [123] 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 papers (P1, P2, P3, and P9) mention the importance of communication across teams, for instance, using the concept of *Communities of Practice (CoP)* [136]. 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, known as "Guilds" within Spotify, their challenges and benefits, have also been studied before, e.g. [124], [125].
- **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 an architect group (TA, see item **A1**) 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 3.5), 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* [32] 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 [61] 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 craftspeople.

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 [137].

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 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 DoD 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.

3.5.4 F Feedback

Feedback loops have always been important in the software industry, as described both by Royce in 1970 [113] and by Brooks (B1) in 1975 [17]. 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 XP [7], and it is intimately tied to the sprint practice of Scrum [9], which also includes explicit review practices.

Lean Software Development [106] 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 3.11 are similar to, or complements, Agile or Lean principles, which is acknowledged in several books, for example, as stated by Mancuso in B9 [83]: "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 develop-

Table 3.11: 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	

ment team, something that also is a crucial trait of Agile (e.g. [7], [9]) and Lean [106] 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 inspiration for the Scrum process [9] 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 the 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 develop-

ment 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 3.12 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

Table 3.12: Elapsed Calendar Days Per Feature Size and Activity.

		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

No QA is the number of features where planned system verification was deemed unnecessary.

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 3.12 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 by the TA and QA groups (see

item **C3**) . While the TA group reviewed the technical solutions, the QA group focused on reviewing test strategies such as test coverage and test structure.

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, 3,802 reviews took place, out of 54,637 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 IP.

  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 TA and QA roles (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* [106].

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]" [17]. 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 3.5 (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).

3.6 Discussion and Implications

3.6.1 The principles and practices of software craftsmanship — in literature and in our case study

Tables 3.6, 3.8, 3.10, and 3.11 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 **C5.2** *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.

3.6.2 What are the consequences of applying the software craftsmanship principles and practices in real life?

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 **C3.1** *Cross-team forums* 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.

3.6.3 Software Craftsmanship vs. Agile Software Development

Following the organization in paper P5 [82], 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.

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* [135]. According to the State of Agile Report [26], 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.*

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. [74] 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.

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 [83].

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.

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.

3.6.4 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 and Poppendieck in [106].

- *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 [106], there are also differences. The program planning and budgeting were performed by different roles in the studied case, outside the scope of this article. 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 article.

3.6.5 Returning to the Software Craftsmanship Manifesto

Looking at the manifesto[13] values through the lens of our anatomy, we find the following:

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

- "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 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 [13]. 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 [91]. 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.

3.7 Validity

In this section, we discuss the threats to validity from four different angles: *construct validity*, *internal validity*, *external validity* and *reliability* [142].

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 [109].

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 [96].

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 [42]. 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 SLR section of the article, 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.

3.8 Conclusions and Future Work

3.8.1 Conclusions

Regarding **RQ1**, how Software Craftsmanship has been conceptualized in literature, although the principles have a long history in gray literature, we found comparatively few published research articles. In our SLR, we could find only

18 papers discussing the principles to some extent, see Table 3.4. 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 3.3 and Table 3.6, 3.8, 3.10 and 3.11.

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.

3.8.2 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 to, and overlapping with, software craftsmanship. Thus, we would like to explore this in detail in subsequent publications.

Chapter 4

The Hidden Cost of Backward Compatibility: When Deprecation Turns into Technical Debt

This chapter is based on the following paper:

A. Sundelin, J. Gonzalez-Huerta, and K. Wnuk, "The hidden cost of backward compatibility: When deprecation turns into technical debt - An experience report," in *Proceedings - 2020 IEEE/ACM International Conference on Technical Debt, TechDebt 2020*, 2020, ISBN: 978-1450379601. DOI: 10.1145/3387906.3388629

Abstract

Context The micro-services architectural pattern advocates for the partitioning of functionality into loosely coupled services, which should be backward compatible, to enable independent upgrades. Deprecation is commonly used as a tool to manage multiple versions of methods or services. However, deprecation carries a cost in that tests might be duplicated and might rely on services that have become deprecated over time.

Objective Using the terms of the Technical Debt metaphor, we explore the consequences of deprecation, and how it has affected the test base during seven years.

Method We take an exploratory approach, reporting on experiences found before and after servicing parts of the incurred Technical Debt. We mine code repositories and validate our findings with experienced developers.

Results We found that the growth of deprecation debt varied a lot. Some services experienced substantial growth, but most did not. Unit tests, where deprecation is visible in the developers' tools, were much less affected than integration tests, which lack such visualization mechanisms. While servicing debt of 121 out of 285 deprecated services, we discovered that up to 29% of the spent effort could be attributed to accrued interest. However, this is an upper bound; there could be less impact, depending on whether scripting could be used to service the debt or not.

Conclusion This paper illustrates that integration tests can be viewed as a debt from the perspective of deprecated services. While the pattern was that deprecated services (debt principal) experienced no or little accrued interest, some, highly used, services experienced a lot, particularly during stressful times. Java-based tests, where deprecation is visible in the IDE, did not experience a similar pattern of increasing debt. We postulate that deprecation debt should be kept visible, either using developer tools or statistical reports.

4.1 Introduction

The current trend in software engineering is to split functionality into small, focused, loosely coupled services, often called microservices [45], using the DevOps moniker [5]. To achieve loose coupling, services should be independently developed. Zimmermann et al. [144] highlight that continuous upgrade-ability and backward compatibility are necessary for achieving loose coupling because they allow clients to upgrade to newer available services at will, irrespective of how the service is upgraded.

However, we have not found any study that explores the converse problem—the importance of removing unused services, and how to visualize whether or not the clients use "the most current" version of a service.

Exposing several versions of the same service carries a cost, both from a development and verification point of view. The test base can be seen as "the first client" of a service, as it is the first code that exercises the service in order to verify the expected behavior.

This paper describes the impact of service deprecation on an existing test base, and the effect that this has had on a product over the course of seven years. We take an exploratory approach and mine the Git repository of the studied product for data related to the exposed services.

A core principle used when developing new versions of the services in the studied system has been backward compatibility. A new version of service should behave identical to the currently existing version, though it might accept other types of data, or return more or less data, based on the given input. Deprecation markers have been used to flag which versions to avoid. The studied system contains one non-deprecated version and N (≥ 0) other deprecated versions of any published service.

The paper is structured as follows: In Section 4.2, we cover the related work in the area. In Section 4.3, we report on the research methodology, and relate to the background of the studied system. In Section 4.4, we report the results found during the study. In Section 4.5, we discuss the threats to validity, and in Section 4.6, we draw conclusions and elaborate on the implications for the research area and industry in general.

4.2 Related Work

Deprecation is a way to signal to API consumers that there are other, more up-to-date ways to perform a given task. Morgenthaler [4], views deprecation as software aging made manifest, and maps this directly to technical debt.

A complication is that very often, deprecated methods or services remain for very long times in APIs. As an example, the Java Standard Library class *java.util.Date* has contained four deprecated constructors and 18 deprecated methods since the release of Java 1.1 in 1997. Today (in Java 12[1]), these deprecated methods and constructors are still present, together with two non-deprecated constructors and 11 non-deprecated public methods.

Sawant et al. [117] studied how developers (API consumers) in open-source projects react to deprecation events in popular Java APIs. The overall conclusion is that developers seldom react at all, and rarely keep up with API evolution. A weakness of this study is that it focuses only on open-source projects. In [118], the same authors, apart from increasing the number of studied projects, also conducted a survey, trying to reach out to developers on proprietary projects. The conclusion in this paper is similar to the prior study, and respondents point to the perceived complexity and time consumption as

[1] https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Date.html

the main reasons for not upgrading to the latest version of the APIs and keep using deprecated versions.

The same finding was made in a study by Kula et al. [73], who investigate the efforts of library migration covering over 4600 GitHub software projects and 2700 library dependencies. Results from this study show that 81.5% of the projects kept their outdated dependencies, citing unawareness, extra effort, and added responsibility.

As of Java 9, the deprecated annotation has been enhanced[2] with an attribute indicating that a construct is deprecated with the intent to remove it in a future version. The motivation for this is that the deprecation marker ended up being used with different purposes, and the theory is that if developers realize that a construct is deprecated with the intent to be removed in a future release, they are more likely to update. As part of this work, there was also additional tools developed, available as part of the Java ecosystem.

Snipes and Ramaswamy [126] proposed a sizing model for managing technical debt related to the deprecation of third-party software components. The model, based on a sigmoid curve, takes into account the lines of code affected, how much (as a percentage) of the API that is deprecated, and the age (in years, with a cutoff proposed to 5 years) of the software component. The authors illustrate the model using Monte-Carlo simulation.

Curtis et al. [34], proposed estimating the principal amount of technical debt based on static code analysis tools and a parameterized formula. The authors illustrate the model by using the tool and formula on 700 applications, comprising 357 MLOC in various languages, from 158 vendors. The article does not state whether only production code was analyzed, or if test code also was included in the metric. A conclusion is that the proposed formula is highly sensitive to the given parameters. The authors illustrate this with three sets of parameter values.

Codabux and Williams [24], studied an organization adopting Agile practices and focused partially on the management of technical debt. Research questions center on characterizations, consequences on the development process, addressing, and prioritization of technical debt. Test debt in the study relates mostly to missing tests and lacking automation of tests.

Kruchten et al. [72], relate to *Testing Debt* in three different ways: "*Imperfection or suboptimal design and coding of tests*", "*Misalignment between the tests and the actual code*" and "*Challenges of SaaS contexts*". Tests, especially when automated, are also code and need to be designed just like production

[2] https://openjdk.java.net/jeps/277

code. When the purpose is unclear, time is spent trying to figure out whether failures originate from the tests or the production code.

4.3 Research Methodology

We take an exploratory approach, utilizing mixed methods and triangulation to elicit experiences from the studied organization. As the studied product has been developed for nine years (of which seven in live operation), a complication is that many developers are no longer available for feedback on conclusions. When possible, we try to confirm and contrast our findings with experienced developers, with a long track record in the product (a handful of which has been with the product since the start). We mined the version control system (Git logs), where all changes to files are stored.

The developers of the studied system took principled decisions related to the support of multiple versions of exposed services:

- New versions are introduced when the existing service cannot accommodate newly requested features, but the core function is the same.
- When introducing a new service version, all old service versions are deprecated. This leads to each service having exactly one version that is "current."
- When a version is deprecated, it remains in the product, and existing test cases continue to use it. New test cases are written for the new version, but old ones are not converted, even if they use the old, deprecated version of the service.
- Removing a service version is done as a planned task, based on extensive communication with impacted stakeholders and requires updates to all systems (including test cases that might use it for other purposes than explicitly testing the service version itself).

4.3.1 Research Questions

Based on the above principles, we formulate the following four research questions:

- **RQ1** How has the decision to avoid converting the existing test base when deprecating a service version contributed to the spread of Technical Debt?

- **RQ2** If there is a contribution to Technical Debt in the test code base, are there any differences between the Java-based files (where deprecation is visible in the IDE) and the non-Java-based text files (where no such feedback is visible)?
- **RQ3** How has the growth in deprecated service version usage contributed to the spread of Technical Debt?
- **RQ4** What is the likely cause of the spreading of deprecation debt?

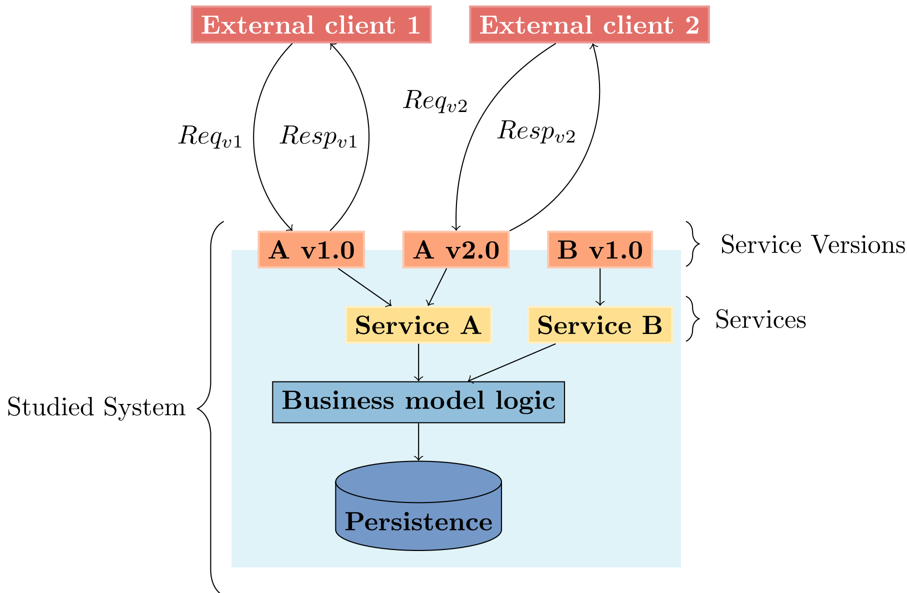
4.3.2 Case Description

The system under study forms part of a FinTech global product that enables access to financial services via mobile phones and the Internet. It is typically installed in a high-availability configuration, with geographical redundancy, to meet service uptime requirements. Starting in 2013, the system today serves tens of millions of users in around 15 installations around the world, each integrating with several hundred different other systems. The system contains several transaction-intensive applications, with incoming and outgoing interfaces and persistence layers in the form of different databases. As it is a financial application, security plays a dominant role.

Figure 4.1 illustrates the system and the role of different services and how each service exposes different versions of requests and responses. Clients external to the system may choose to use any of the exposed service versions to invoke a particular service.

The studied system was developed with test automation in mind, using different test strategies, such as unit testing, integration testing using public APIs, and system testing by a different organization than the development teams. The development has been conducted in an Agile manner, with frequent releases, and incorporating core Agile practices, such as continuous integration and as short feedback loops as is practical. Continuous Integration tools such as Jenkins, giving fast feedback to developers on the quality of their product, has been used during the whole development period.

The system exposes a set of HTTP-based services to the surrounding systems, using XML for marshalling and unmarshalling. As the number of clients to the system is essentially unbounded (or at least unknown at design time), backward compatibility on the protocol level has been an important factor in the design of the system. For this purpose, protocol schemas were developed using XML Schema (XSD), and the system developers assumed that at least some clients, possibly also security firewalls, would enforce strict validation of the



(a) External client 2 uses service A in version v2.0 (request Req_{v2} and response $Resp_{v2}$) External client 1 has not yet upgraded, and is still using v1.0.

Figure 4.1: Schematic view of the studied system, exposing services in different versions.

request and response messages against the corresponding schema. Due to the compatibility principle, the number of versions of the same service has grown over the years. Another principle has been that when developing a new version of a service, all older versions are marked as deprecated, using the regular Java deprecation mechanism (annotation and Javadoc). This information was included in the product API documentation, which was generated from Javadoc.

As the system has been developed in Java, deprecation warnings appear integrated into the IDE, such as Eclipse and IntelliJ IDEA. However, integration test cases, interacting with the services via officially supported interfaces, such as XML requests over HTTP, has been developed in a text-based language, custom written for the application. This language is also specified in XML and is executed via a custom runner.

The use of XML text is similar to text-based BDD[3] languages, such as Gherkin, but has proved hard to sustain for the 9000 unique test cases. Current developers are unwilling to modify existing test cases, and express frustration at the lack of IDE support for working with plain-text-based languages. This is in contrast to the comparatively well-developed search and refactoring support in IDEs such as Eclipse or IntelliJ IDEA. As the XML language lacks the deprecation mechanism that is available in plain Java, there is no warning when using a deprecated version of a service. Thus, the developer writing or maintaining test cases lack feedback on whether or not this should be changed.

Figure 4.2 shows the evolution of the number of files of production code (*prod*) and tests over the studied period. The combination of *int.tc* and *int.setup* files contains the integration test files, with *int.setup* referring to the code used for setup, tear-down, and utility functions used in integration test cases, whereas *int.tc* refers to the actual code of the integration test cases. *Unit* refers to the Java-based test code, typically in the form of unit tests.

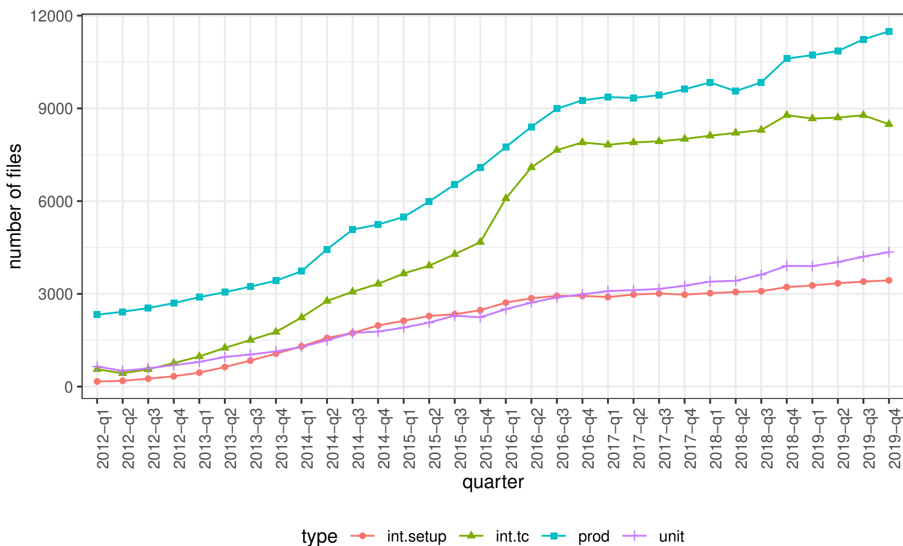


Figure 4.2: Production and test files per quarter

[3]https://dannorth.net/introducing-bdd/

Table 4.1: Average lines per file across quarters.

Type	$\bar{x}$	σ	$\hat{x}$	first	last	min	max
prod	73.4	3.6	73.8	74.2	88.4	66.8	88.4
unittest	172.3	24.4	181.8	139.6	234.5	133.1	234.5
int.tc	155.0	10.3	154.8	187.2	148.5	135.4	187.1
int.setup	83.4	23.1	82.8	117.8	61.5	61.5	135.6

The Java-based files, both production and unit tests, show steady growth over the studied quarters. Employing standard linear regression, we see that production files grow with 330 files/quarter, p-value less than $2 * 10^{-16}$, and adjusted R^2 of 0.97. Likewise, the unit tests grow with 128 files/quarter, p-value less than $2 * 10^{-16}$, and adjusted R^2 of 0.99.

For the integration test cases (*int.tc*), the picture is somewhat different. The period up to and including Q4 2015 exhibit a growth rate of 300 files each quarter, with a p-value of less than $8 * 10^{-13}$ and an adjusted R^2 of 0.97. Then, the ratio increases, between Q4 2015 and Q4 2016, the growth rate is 801 test-case files, with a p-value of less than 0.010 and adjusted R^2 of 0.89. As of Q1 2017, the growth rate has flattened, and is now 84 files per quarter, with a p-value of $2 * 10^{-5}$ and adjusted R^2 of 0.80. The last quarter of 2019 shows a decrease in test case files, which reflects the fact that one developer removed, in his opinion, unnecessary integration tests, by converting some to unit tests and removing others altogether, in a refactoring operation. For the integration test setup files and shared functions (*int.setup*), the picture is slightly different. Up until and including Q2 2016, the setup files were growing with 175 files/quarter, p-value less than $3 * 10^{-16}$ and adjusted R^2 of 0.98. As of Q2 2016, the growth stagnates to 41 files/quarter, p-value less than $4 * 10^{-8}$ and adjusted R^2 of 0.90.

When discussing efforts in terms of counts of files, it is important to note that each file can grow as well. Table 4.1 contains data regarding the average lines per file type, and its variation across the quarters. For all four file types, the mean ($\bar{x}$) and median ($\hat{x}$) are quite similar, and the standard deviation is also quite small. The *first* column refers to Q1 2012, and the *last* to Q4 2019 (before servicing the debt). Relative to their size, the setup files have the largest standard deviation. Both integration test cases and setup files have a downward trend in file size across the quarters, while the Java-based files (production code and unit tests) are slightly growing, unit tests more so than production code.

The number of authors per year has varied somewhat, as illustrated in table 4.2, where the number of unique authors, as identified by the Git *Author:*

Table 4.2: Average number of authors each quarter, per year

Year	$\bar{x}$	σ	$\hat{x}$
2011	26.00	7.9	28.0
2012	36.25	2.1	36.0
2013	42.25	6.2	40.5
2014	50.75	1.3	51.0
2015	58.00	3.7	58.5
2016	86.00	5.0	86.5
2017	25.00	10.7	20.0
2018	41.50	15.2	47.5
2019	46.75	3.9	46.5

tag, each quarter is summarized (4 quarters per year). After steady growth until 2016, the last year explosively, there was a contraction during 2017, followed by slower growth until the end of the study period.

4.4 Results

4.4.1 The origin of deprecation debt

When faced with changed or additional requirements related to an existing service, developers face a dilemma:

Update the existing service, to accommodate new requests or response parameters, or change existing parameters. An equivalent solution is to delete the old service, and introduce a new one with the same name. No Technical Debt would be associated with this option.

Keep the old service version and introduce a new version of the same service, where the required changes are made. The old service version is deprecated, and is associated with a Technical Debt principal, as it has to be maintained alongside the new version, which solves a similar business purpose (being the same service). The consequences of keeping the deprecated version are not only visible at code level, but might be even more severe at the testing level since the integration tests, as will be illustrated later, can keep using the old versions of the service. The propagation effects

of deprecation to the test cases are one example of a code TD-item that appears as a TD-item in the testware [2].

Either of the two options can be exercised, and for the studied system, the favored solution was **Update**, unless this was prevented by protocol conventions. Reasons for being unable to choose **Update** were such that would cause existing clients to break, for instance:

- Changing the XML Schema type of an existing parameter (e.g. `xsd:integer` to `xsd:string`), in either the request or the response.
- Adding a new mandatory request parameter P, as this would break old clients who would not send P in their requests towards the system.
- Making an existing optional request parameter P mandatory, as existing clients would be unaware that P was required.
- Adding an optional or mandatory parameter in a response, as this would break those clients who would perform strict schema validation on received responses.

Some of the reasons for choosing option **Update** were:

- If the existing service had not yet been part of any product release, so external clients would not have had the opportunity to interact with it, or learn about the API.
- Adding an optional parameter P to a request, as existing clients could refrain from sending P.
- Refraining from returning an optional parameter P in a response, as existing clients would already have had to deal with the absence of P when interpreting the response.

Initially, the system had no monitoring of which version of a service is currently used by which customer. This monitoring was introduced in 2017. During late 2019, there were efforts made to clean up the unused, deprecated service versions. Based on usage data, 121 deprecated service versions were identified and removed. At the end of the studied period (before cleanup), the system comprised 632 unique services, which exposed 891 different versions of services. In other words, almost three out of ten (29%) of the service versions were deprecated at the end of the studied period.

Kruchten et al. [72] refer to debt principal as proportional to the effort that a development team would expend to eliminate it. The interest incurred by a technical debt item is the additional effort needed to eliminate the debt if the item is left in the system. This view is also shared by others [4]. For our deprecated service versions, the principal would include the effort needed to remove the integration test cases (*int.setup* and *int.tc*), plus the production code (*prod*) and unit tests (*unit*) maintenance between the introduction of the usage statistics and the pay-back in Q4 2019. We classify our deprecated service versions as two different types of Technical Debt Items (TD-items):

- **Type I TD-items:** deprecated versions of services that the usage metrics reported as not in use in any external customer installation.
- **Type II TD-items:** versions of services, that, although deprecated, were reported as in use in some external customer installations by the usage metrics.

While Type I TD-items require a relatively small effort to remove the service versions (relatively low principal), the second Type II TD-items requires more coordination with the customer adaptation teams, for them to adapt the customer installation to the ultimate versions of the service before the removal of the deprecated service versions both from the code and test base.

We treat production code and unit tests similarly, as they are both specified in Java, a statically typed language. Likewise, we count both integration test cases and their setup files together, as they both are specified in untyped XML.

$$javacode = prod + unittest$$

$$int.test = int.tc + int.setup$$

We will use the following set of equations to calculate the usage of a particular service version v at quarter q in files of type t:

$$FILES(q,t) = \{f : quarter = q, type(f) = t\} \tag{4.1}$$

$$PRESENT(v,f) = \begin{cases} 1, & \text{if } v \text{ occurs in file } f \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

$$COUNT(v,q,t) = \sum_{f \in FILES(q,t)} PRESENT(v,f) \tag{4.3}$$

We define $COUNT(v, q, t)$ as the number of files of type t in which service version v was present in quarter q. We choose the number of files rather than the number of occurrences with the hypothesis that the marginal cost of changing an additional occurrence, once a file has been found, is negligible.

To calculate the distribution of service versions throughout the different kinds of files, we select quarter Q4 2019 (before servicing the deprecation debt), and enumerate all the 891 service versions. For each service version, we calculate the $COUNT(v, 2019Q4, t)$, values, where t is either Java (*java*) or XML files (*int.test*). The result is plotted in figure 4.3, as a histogram with 30 logarithmic bins.

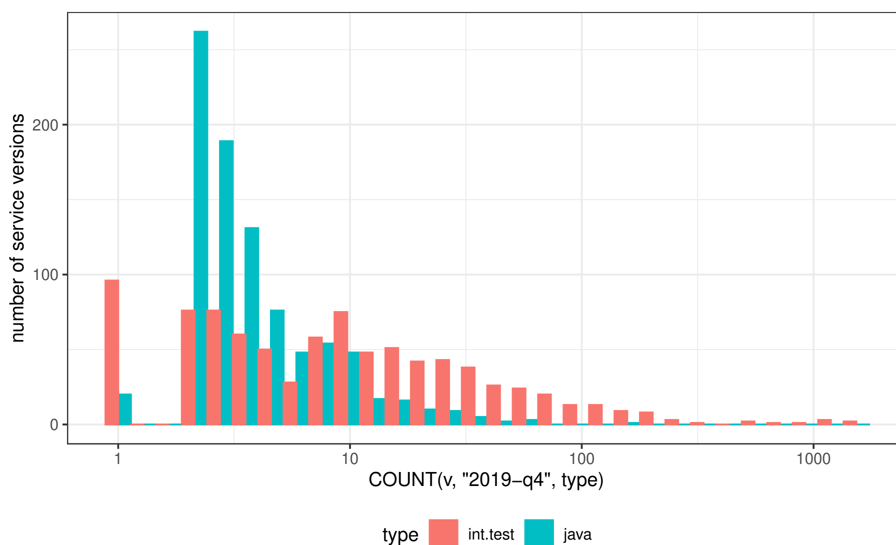


Figure 4.3: Occurrences of services in Java or integration test code

Note the heavily right-skewed distribution, where relatively few service versions are heavily used, while the majority of service versions are used in only a handful of files. Occurrences in Java-based files occur more seldom (closer to the y-axis) than integration tests, though 96 service versions are present in only one integration test file. This is also visible in table 4.3, where the Java-based code have a considerably smaller mean ($\bar{x}$) and median ($\hat{x}$) than the integration tests. Table 4.3 also displays the number (N) of different service versions (noting that

Table 4.3: Occurrences in files.

Type	N	$\bar{x}$	σ	$\hat{x}$	$Q_{90\%}$	$Q_{95\%}$	**max**
javacode	891	5.3	7.9	3	10	15	165
int.test	867	31.0	110.0	8	57.4	106	1529

24 service versions that are used in Java code are not used in integration tests), standard deviation (σ), 90th and 95th percentiles, and the max value.

Production code and Java-based tests, such as unit tests, typically use a broader spectrum of interfaces, such as internal domain-model interfaces, whereas the primary purpose of the integration tests is to exercise and validate the external application interface (that is, the exposed services, in their different versions). The most heavily used service version appears in 1529 integration test files (including setup files), out of around 12300 files.

4.4.2 Deprecated services

All analyzed services start as "non-deprecated," in version 1.0, but some will evolve together with the product to later versions.

Result: Time series of usage data for each deprecated service version following deprecation
$end_of_study \leftarrow Q4_2019$;
for $v \leftarrow all_deprecated_service_versions$ **do**
  $when \leftarrow find_deprecation_timestamp(v)$;
  $following_quarter \leftarrow next_quarter(when)$;
  for $q \leftarrow following_quarter$ **to** end_of_study **do**
    $count[v, q, Java] \leftarrow COUNT(v, q, Java)$;
    $count[v, q, XML] \leftarrow COUNT(v, q, XML)$;
  end
end
return *count*

Algorithm 1: Calculating usage of deprecated service versions

Algorithm 1, illustrates the process to collect the usage data for every deprecated service version, starting at the quarter following the deprecation event as recorded in the Git version control log. Thus, the initial data point for a service version closely reflects the state at the time when it was deprecated. In the ideal

world, each service version should show a flat or decreasing line, as its usage decreases as the product evolves. It is important to note that also deprecated service versions, which are still in use and delivered with the product, should have some usage in tests (the target is not zero files). However, typically this number should be relatively low (in the single digits), as the officially supported version should be used instead.

Figure 4.4 depicts the situation for the six most frequently used deprecated service versions[4], and figure 4.5 for the following six. All of these occur most frequently in integration tests. A few service versions are fairly flat, a few show moderate growth, and a few show extreme growth up until Q2 2016. Following the end of 2016, most of the usage is flat, and there are few changes in usage.

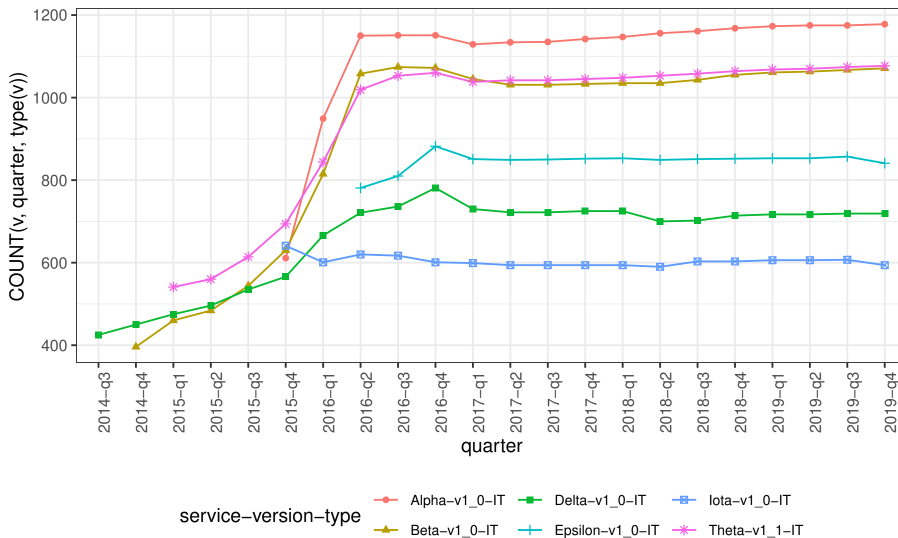


Figure 4.4: Top 6 used deprecated service versions per quarter

The situation in the six deprecated services most frequently used in Java files is depicted in figure 4.6. While there is a slight increase for some services, in general the situation is much more stable than for the integration tests, and the number of affected files are also much lower.

[4]The names and nature of the services are obfuscated due to confidentiality and security reasons.

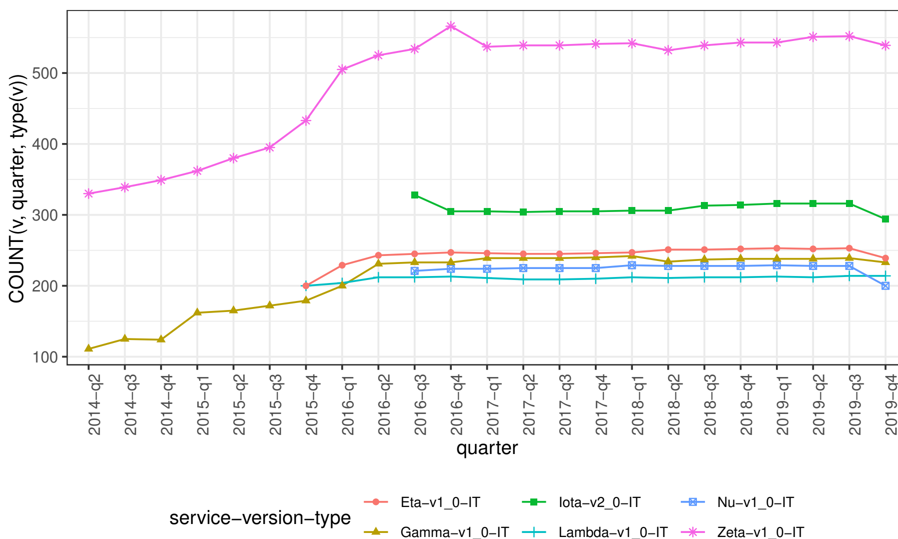


Figure 4.5: Top 7-12 used deprecated service versions per quarter

We note that some services (e.g., *Epsilon*, *Eta*, *Iota*) appear both among the top 12 integration tests and in the top six Java-based files. These are "core services," central to the application at hand. Furthermore, we note that one service, *Iota*, in different versions, is both the sixth and the eighth most used in integration tests.

Most of the quarters, there is no change in the usage of the deprecated service versions. Out of the 7849 data points (one deprecated service version in one quarter is a data point), growth is flat in 6584, i.e. given a random service in a particular version, and a random quarter following the deprecation of that service version, the chance is nearly 84% that there has been no change in usage. The odds for decreasing usage is 5.1% (413 cases), and increasing is 10.9% (852 cases).

Looking at the quarters with the most growth in absolute terms, we see the same three quarters as in figure 4.4. The top 10 growth operations and quarters are illustrated in table 4.4. In the table, the service versions *Alpha v1.0* and *Zeta v1.0* is marked in bold, as being Type I TD-items (deprecated and not being used in live operation). The column *Count* contains the $COUNT(v, q, int.test)$

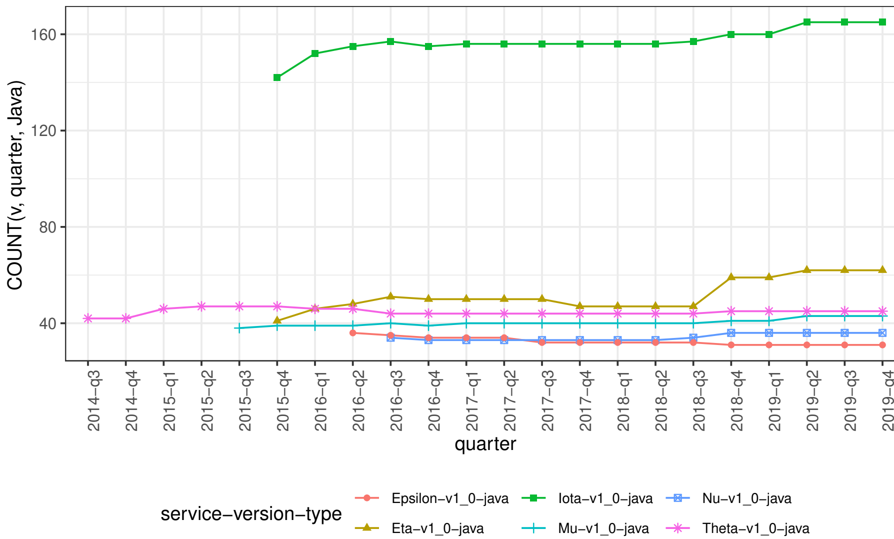


Figure 4.6: Top 6 used deprecated service versions in Java files

metric for the specified service version and quarter, while the *Growth* column, contains the change since the previous quarter. The *Dupl* column is the absolute number of duplicated files of the *Growth*, and the *% Dupl* is the relative percentage of duplicates, compared to the *Growth*. We see that a small number of service versions cause most of the growth in debt. Furthermore, this debt is concentrated to three quarters, spanning Q4 2015 to Q2 2016. Inspecting the *% Dupl* column, we see that for some service versions (*Beta v1.0*, *Theta v1.0*), over half of the added files are duplicates of each other. This indicates that the developers were unaware of a feature that was added to the test execution runner in Q4 2015, where a "shared function repository" could be defined. All of the duplicated files except one are setup files or "utilities" for the test cases.

In figure 4.7, we plot the relative number of file duplicates across each quarter, and the lines of code in these files. We see that as of 2014, around 15% of the files are duplicated, though these only make up of around 5% of the total code base.

In table 4.5, we instead look at the service versions and quarters where the most deprecation debt has been repaid (in the form of files that no longer are

Table 4.4: Most growing deprecated service versions, per quarter.

Service	Quarter	Count	Growth	Dupl	% Dupl
***Alpha v1_0**	Q1 2016	949	338	66	19.5%
Beta v1_0	Q2 2016	1058	243	77	31.6%
***Alpha v1_0**	Q2 2016	1150	201	74	36.8%
Beta v1_0	Q1 2016	815	185	65	35.1%
Theta v1_1	Q2 2016	1019	175	65	37.1%
Theta v1_1	Q1 2016	844	150	48	32.0%
Delta v1_0	Q1 2016	666	100	0	0.0%
Beta v1_0	Q4 2015	630	86	53	61.6%
Theta v1_1	Q4 2015	694	80	42	52.5%
***Zeta v1_0**	Q1 2016	505	72	0	0.0%

Count is the $COUNT(v, q, t)$ metric.
Growth is the change in *Count* since the prior quarter.
Dupl is the number of duplicates among the *Growth*.
% Dupl is the relative number of duplicates.
Type-I TD-items in bold.

used). We see that five out of the top 10 service versions were repaid during Q4 2019. Further analysis of the version history logs revealed that this is related to the prior mentioned refactoring illustrated in figure 4.2, where one developer removed many integration tests, converting some of them to unit tests.

In order to conclude as to how the duplicated files shown in table 4.4 were introduced, we analyzed the version control logs. The analysis indicated that the files, to a large extent, originated from experienced developers. All in all, 102 commits were affected, authored by 25 different developers, and the top 5 contributors caused 73.5% of the duplicates. These top 5 contributors all had several years of experience in the product at the time of introducing the duplication.

Figure 4.8 illustrates the concept of principal, which, for Type-I TD-items, are proportional to the usage in the code base, and interest, which is proportional to the increased usage. The *First* metric is the usage at the quarter following deprecation (different quarter for each service version), and the *Last* metric is the usage at the end of the study, before servicing the debt. Usage before deprecation is not counted as principal by this metric.

In table 4.6, the $\sum First$ column is the sum of all the *First* values for all N deprecated service versions and the $\sum Last$ column is the sum of all the *Last* values. The $\sum Add$ column totals the net additions, and the $\sum Del$ column the net removals. In case one file contains multiple instances of the same deprecated service, it is only counted once, but in case it contains several

Table 4.5: Least growing deprecated services, per quarter.

Service	**Quarter**	**Count**	**Growth**
Nu v1_0	Q4 2019	200	-28
Psi v2_0	Q4 2019	44	-28
Chi v1_0	Q4 2019	20	-28
***Zeta v1_0**	Q1 2017	537	-29
Epsilon v1_0	Q1 2017	851	-31
Iota v1_0	Q1 2016	601	-40
Delta v1_0	Q1 2017	730	-51
Zeta v1_7	Q3 2018	10	-60
Phi v1_0	Q4 2019	21	-105
***Zeta v1_4**	Q4 2019	36	-148

Count is the $COUNT(v, q, t)$ metric.
Growth is the change in *Count* since the prior quarter.
Type-I TD-items in bold.

Table 4.6: Affected files for all deprecated services

Type	N	$\sum First$	$\sum Last$	$\sum Add$	$\sum Del$
javafiles	285	1620	1878	292	-34
int.test	282	10619	13526	3819	-912
Sum		12239	15404	4111	-946

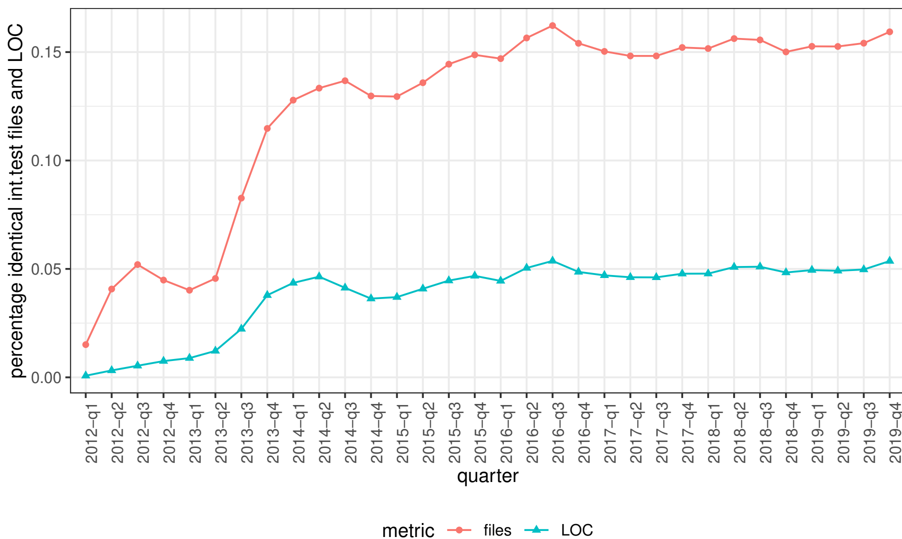


Figure 4.7: Percentage of identical integration test files and LOC

deprecated services, it is counted once per such service. This uses the hypothesis that changing several instances of the same pattern in a single file incurs a relatively minor additional cost, compared to finding the file and changing in one place.

The cost of removing a service version V_{depr}, which has a non-deprecated version V_{actual} can be broken down into the following components:

$$C_{actual}(V_{depr}) = C_{internal}(V_{depr}) + C_{external}(V_{depr}) \tag{4.4}$$

$$C_{internal}(V_{depr}) = C_{removal}(V_{depr}) + C_{update}(V_{depr}, V_{actual}) \tag{4.5}$$

where

- $C_{actual}(V)$ is the actual cost of removing V.
- $C_{external}(V)$ is the cost of removing V by integrators and other clients (outside the development organization).

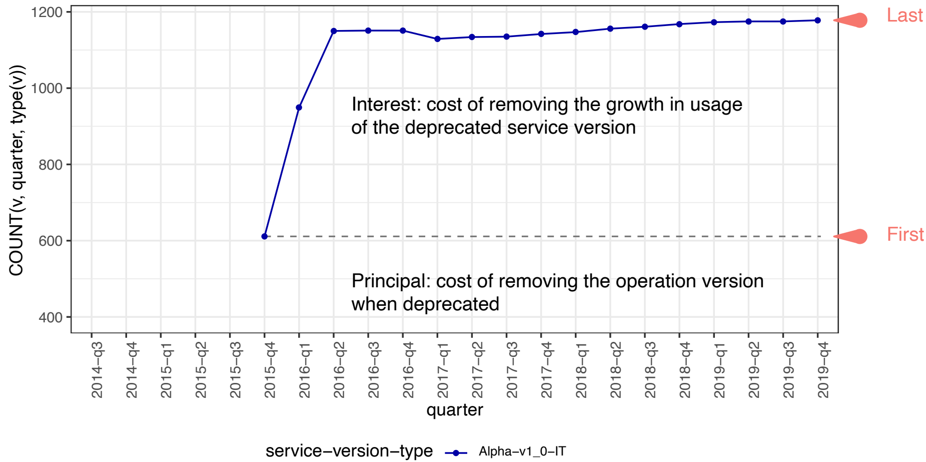


Figure 4.8: Schematic illustration of Principal, Interest, First and Last metric for integration tests of service Alpha v1.0

- $C_{internal}(V)$ is the cost of removing V that is internal to the development organization (code base, including tests).

- $C_{removal}(V)$ is the cost of removing only V and its associated test cases.

- $C_{update}(V_1, V_2)$ is the cost of updating all the remaining tests from V_1 to V_2.

Using the Technical Debt metaphor, the $C_{actual}(V)$ at the time of deprecation can be considered as the principal, as the alternative would be to simply remove V, replacing it with the updated version. We can estimate $C_{update}(V_1, V_2)$ as the additional usage of V_1, not related to the normal tests of this version. In integration tests, it is common that certain operations are used for setting up the data in order to test other services, such as when a login service is used in several integration tests across the test base. This is what we call *excessive usage* of a service version, as opposed to the *expected usage* of a service version which is its source code, its unit tests and integration tests of the service version itself.

Table 4.7: Affected files for unused services (Type-I TD items).

Type	N	$\sum First$	$\sum Last$	$\sum Add$	$\sum Del$
javafiles	121	505	598	105	-8
int.test	120	2383	3428	1313	-268
Sum		2888	4026	1418	-276

For Type-I TD-items, where the $C_{external}$ is zero, we find that the $TD_principal$ is proportional to the usage at the time of deprecation, that is, $\sum First$. The accrued interest is proportional to the growth in usage, that is, to $\sum Last - \sum First$.

$$TD_principal_I \propto \sum First$$

$$TD_interest_I \propto \sum Last - \sum First$$

Thus, the relative growth of $TD_interest_I$ for Java files is $\sum Last / \sum First - 1$, that is $(598/505) - 1$, or 18%. For the integration tests, the corresponding number is $(3428/2383) - 1$, or 44%. Thus, the interest rate for integration tests is much higher than for Java-based tests. We also note that there have been some decreases. In total, 268 files have been corrected for the 120 different integration test service versions, but this has not been able to keep up with the growth.

For Type-II TD-items, the situation is somewhat more complicated, as one also has to consider the lack of test cases as a debt principal, what Kruchten et al.[72] refers to as "*Misalignment between tests and code.*" We note that there are three service versions not tested in integration tests (Table 4.6, N for *int.test*), and one of these was among the Type-I TD items (Table 4.7).

4.4.3 Servicing the debt

We removed 121 deprecated service versions in late 2019, see table 4.7. We note that one of the removed service versions was not present in any integration tests (closer inspection revealed that those tests had been migrated to the new version at the time of deprecation, in effect leaving the old version untested for five years).

Based on data from 21 of the operations that were not part of any other tests, we found that on average, a service version occurred in 4.4 Java files and 3.6

integration test files. This can be used to estimate the $EXPECTED_USAGE$, using the values from table 4.7.

$$EXPECTED_USAGE_{java} = 4.4 * 121 = 532 files$$

$$EXPECTED_USAGE_{int.test} = 3.6 * 120 = 432 files$$

Based on these values, we find that out of the removed operations, there was little, if any, excessive usage amongst the Java files. However, there had been some growth due to interest (93 additional files being affected). The majority of excessive usage can be attributed to the integration tests, which also contributed the most of the additional interest.

The removal of the 121 service versions was carried out as a low-priority task during several months by three developers, with between 4 to 25 years of experience in the industry. The time spent (based on Git logs and estimations by the developers, as this time was not separately time reported) was 160 hours. Part of this time was due to back-porting the changes to an older, but still alive, branch, which proved to be almost as costly as doing the change in the original branch. Given this, it is reasonable to estimate that, had there been no additional growth in technical debt (such as the one during 2016 illustrated in figure 4.4), the removal would have taken between 0% (in case all file changes could have been automated, e.g. via scripts), and 29% ($1 - 2888/4026$) less time, i.e. 114 hours rather than 160. In practice, the truth is somewhere in between, as some changes could be highly automated, and some required more manual actions. These kinds of tasks are also highly dependent on the skill of the developer doing the change, in particular as the plain-text-based XML language lacks IDE support for refactoring operations such as renaming methods. All three developers were well versed in the Git version control system, scripting tools, and regular expressions, and used them, together with the test base and the Continuous Integration environment to verify system behavior before and after each change.

4.4.4 Commit counts

It could be argued, from a theoretical standpoint, that a file that is untouched and never read has no technical debt, even if it contains TD-items (in our case deprecated services). In figure 4.9 we visualize the activity on the top six deprecated service versions, starting from the quarter following deprecation, plotting how many commits per quarter affect files where these services are used.

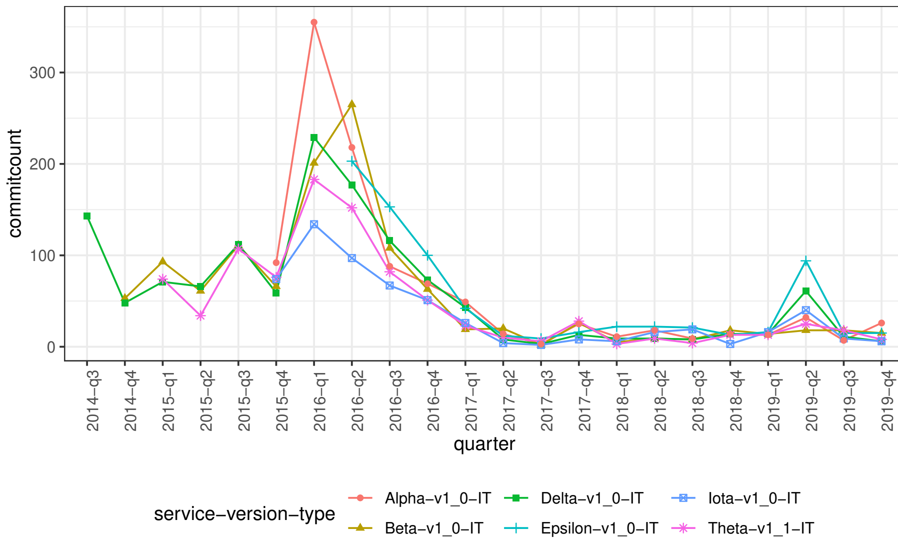


Figure 4.9: Count of commits affecting files using the top six deprecated service versions across each quarter

We see the many commits during late 2015 and 2016, where most of the commits were introducing the deprecated services into the test base. Then, a low period during 2017 and 2018, followed by high activity during Q2 2019, where many commits were touching files related to the top deprecated services. During this quarter, two new requirements were developed, adding configurability where previously more rigid business models had been used. In total, 78% of the commits for the ten most used deprecated services during Q2 2019 were related to these two requirements. Thus, figure 4.9 illustrates that for almost two years, there was not much movement in the files related to the top deprecated services. Then, during early 2019, two requirements were to be implemented, which caused much effort in maintaining these files (i.e., interest that could have been avoided if these deprecated versions would have been removed). This illustrates the highly non-linear nature of technical debt management, making it even more important to visualize it so that efforts to mitigate it can be estimated appropriately.

4.4.5 Discussion

Regarding **RQ 1**, whether the rule to avoid updating existing tests contributed to the spread of Technical Debt, we experienced an uneven distribution of deprecation debt among the deprecated services. Since a small number of highly used services caused a significant increase in debt in integration tests, we hypothesize that highly used services should be treated differently than others.

Regarding **RQ 2**, whether the XML-based integration tests and the Java-based files show any difference in technical debt, we experienced very little debt for unit test cases since the number of affected files remains small. The integration tests show large growth in debt, originating from a small amount of highly used services, as shown in figure 4.4 and figure 4.5. After 2016, we experience a slower growth of integration test cases in general, and the product appears to have been tested using other means (such as Java-based tests), with minimal usage of the integration test framework.

Regarding **RQ 3**, how the growth in deprecated service version usage has contributed to Technical Debt, we note that most of the growth in deprecated services usage occurred during a limited time, between late 2015 and mid-2016. Integration tests were much more affected than Java-based files. For the services removed in late 2019, the accrued interest in the integration tests was calculated to 44%, versus 18% for the Java-based files. We have identified considerable duplication among the integration tests, particularly the setup files. Undue duplication incurs technical debt, even if a particular service is not deprecated, but in this case if a service is deprecated but still used in integration tests might lead to unexpected tests results due to the combination between deprecated and non-deprecated services. Thus, the general higher frequency of copied code in the integration tests could explain why there were more duplicates of deprecated service versions.

On a positive side, we note some repayment of interest for both types of files, though the decrease does not weigh up the increase in debt.

Regarding **RQ 4**, about the likely cause of the spreading Technical Debt, we can identify at least three causes of the spreading of deprecation debt:

- Backward compatibility causes technical debt unless the protocol can accommodate the needed changes without breaking older clients. Depending on the strictness of validation in clients or intermediate systems, different amount of debt is incurred. In the studied system, the rigid protocol rules, and strict security requirements (schema validation in firewalls) contributed to the large number of deprecated service versions.

- By examining the integration tests, we could see a pattern of copied setup files. Even though the integration test engine had the possibility of using a "shared function repository" since Q4 2015, this feature was not widely used during early 2016. Since 2014, the level of duplication in the integration test base has been around 15% on a file-by-file basis, comprising approximately 5% of the codebase, as shown in figure 4.7.

- Regarding the period with the most growth, late 2015 to mid-2016, we could not see direct causes due to the growth in personnel during this time, though indirect causes, such as stress on experienced developers, can not be ruled out. Feedback from three developers with knowledge of the situation unanimously report this time as highly challenging (meeting a customer deadline). This is consistent with what is reported by Kruchten et al. [72]: "The most likely cause [of Technical Debt] we have observed is schedule pressure".

To summarize, we state a hypothesis that if an appropriate visualization had been in place for the integration tests, it is likely that both the duplicated files and the added deprecation debt could have been avoided during the stressful times of 2015-6. This might have saved up to 29% of the removal effort in late 2019. It is important to note that the distribution of services is highly right-skewed, meaning that measures of central tendency such as the mean, median, and standard deviation values are not very helpful. Instead, "top-N", or "most-growing" lists should be used to keep track of the code.

4.5 Threats to validity

We discuss the threats to validity from four different angles: *construct validity*, *internal validity*, *external validity* and *reliability*, following guidelines outlined in [141].

Construct validity deals with whether the studied measures reflect what the researcher has in mind, and what is stated in the research questions.

In this study, we use version control data (source code and revision history information) to draw conclusions. A threat to validity to this approach is that "you only see what was built," which is a form of survivor bias — i.e., you might see what was built, but not how or why it was built that way. To increase construct validity, we presented our conclusions to five of the remaining developers, validating our assumptions about why certain solutions were used.

When using version control logs as a data source, an important aspect is to consider the branching pattern and whether or not the studied organization commonly used rebasing commits. The Git version control system allows authors to “squash” commits, which may have been performed by different authors, at different times, into one new commit, discarding the constituent commits. For the studied system, this was not an approved practice, as the organization valued to see each commit, as it was written and pushed to the central repository. Most of the development took place in a single “master” branch for the duration of the study. Features were developed in other branches and later introduced into the master branch, typically via the Git rebase function, which keeps a linear history by rewriting commits, preserving author information and commit dates.

Internal validity deals with whether there might be other, non-studied factors that could explain some of the findings. We have gathered quantitative data (Git logs, usage statistics), but also the Git ways of working for the studied system, which allows us to study the phenomena (i.e., the effects of deprecation on TD and its spread). In addition, to validate our conclusions, increasing internal validity, we triangulated our data by providing our conclusions and feedback to five developers of the system, one of which was the author of much duplicates during 2015-2016, and asked about their opinions. This provided valuable insight into the conditions and the time pressure experienced by the development organization.

External validity concerns to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside of the investigated case.

This paper is about experiences from a particular system, with a particular toolset. We have tried to describe characteristics that might enable others to judge whether the findings are applicable for other systems, but we cannot claim generalizability across all possible systems or organizations.

Reliability concerns whether or not the data and analysis are dependent on the specific researchers. Most of the data in this report are collected from quantitative sources, such as Git logs, and processed and visualized using standard statistical tools. As such, there is little room for bias in the processing of the collected data.

Interpretation of the processed data runs the risk of introducing reliability threats. We strove to reduce reliability threats by frequent interactions with the studied organization, especially with the developers that had been part of the product development team during the whole studied period, and elicited feedback from five of them.

4.6 Conclusions

As illustrated in this paper, to combat deprecation debt, it helps to keep it visible. Many IDEs today show deprecated classes in a different font (e.g., strikethrough), and this could be one reason why the unit tests do not show the same growth as the integration tests.

Another finding is the uneven distribution of the contribution to the debt. For the majority of services, the decision not to update deprecated usages in test cases did not spread any technical debt. However, the decision not to update some highly used "core services" caused up to 29% of increased effort in converting the test base at service removal. The addition of this debt mostly occurred during three quarters, between Q4 2015 and Q2 2016, a time that was identified as particularly stressful, which aligns with the findings of Kruchten et al. [72].

After the downsizing of the product, the integration tests were mostly untouched, as the new developers valued other test principles, such as Java-based testing (relegating much of the prior testing to unit tests). This suggests that as the test base grows, so does the importance of the IDE support (refactorings, static code analysis, duplication detection).

The usage statistics turned out to be a valuable tool to identify unused service versions (Type-I TD-items), supporting the removal of these.

Chapter 5

Dear Lone Cowboy Programmer - your days are numbered!

This chapter is based on the following paper:

A. Sundelin, J. Gonzalez-Huerta, K. Wnuk, *et al.*, “Dear Lone Cowboy Programmer - your days are numbered!” *Communications of the ACM*, Submitted 2021-07-22

Abstract

Since its inception, software development has been recognized as a highly technical activity, where, at times, highly skilled professionals have been tempted to face technical problems on their own. In the past, software developers, may have been inclined to create solutions as if they were the only ones who needed to understand the solutions. However, nowadays, the disciplines of software development and systems development have undergone significant change. Current software development requires more *crafting* skills, in addition to engineering skills. The lone-cowboy programmer will soon have no place in properly organised software development projects. Current practices demand that a productive programmer be tasked to develop both *working software* (as claimed in the Agile Manifesto) and *well crafted* software. Accountability, pride in one’s work,

continuous learning and mentorship are characteristics of the profession that we should promote if we want to enable an attitude of *craftsmanship* within software development. This paper provides experiences of craftsmanship, and argues why software craftsmanship is good for the practitioner and software development organizations. To support this claim, we have analysed the development of a product that was developed by following several craftsmanship principles. We observed the product's development for seven years, and interviewed several professionals who were involved in its development.

5.1 Introduction

Software engineering emerged as a professional practice in the late 1960s as a reaction to the "software crafting" era, where the "code and fix" approach was prevalent [13]. At this time, a "hacker culture" fostered a culture of "cowboy programmers," who could hastily patch together something during an "all-nighter" to meet an important deadline. However, the efforts of these cowboy programmers often led to code that was unmaintainable and often confusing.

Today, programmers enjoy access to tools that can be used to develop and manage software that their 1960s counterparts could only dream about. Efficiency gains and the standardization of hardware, operating systems, and development and collaboration tools have tremendously increased the development potential of a single developer. This could be one reason why the popularity of the "lonely programmer" mentality remains surprisingly prevalent among software developers and engineers.

The advent of Agile Software Development and its focus on rapid software delivery has perhaps reinforced the desirability of remaining a "cowboy programmer" for some who may even be willing to sacrificing quality aspects for the sake of fast delivery. The lack of a long-term perspective, and a disregard for the skills needed to stay productive over time, appear to be two of the main challenges facing those that strive to introduce the concept and practice of agility to software development organizations.

The software industry's level of adoption of Agile Manifesto principles, in particular its perceived lack of focus on the more technical practices, prompted the formulation of the Manifesto for Software Craftsmanship[1] in 2009. The Software Craftsmanship Manifesto incorporates the following values:

- Not only working software, but also well-crafted software.

[1] https://manifesto.softwarecraftsmanship.org/

- Not only responding to change, but also steadily adding value.
- Not only individuals and interactions, but also a community of professionals.
- Not only customer collaboration, but also productive partnerships.

Several scholars have commented on software craftsmanship. For example, Jacobson [63] has argued that engineering is a craft that is supported by theory, while Bergtröm and Blackwell [10] argue that professional practice is "craftwork." In their empirical study, Lingel and Regan [81] derived different conceptualizations of the concept of "craft" in building software by using a sample of twelve participants, collecting subjective opinions via interviews and a focus group. Lucena and Tizzei [82], on the other hand, present a so-called "experience report" (written from the perspective of a team member) on a Scrum project that applied the craftsmanship principles.

For the present study, we followed a project aimed at developing a financial industry product over seven years. Already at the inception of the project, the developers were inspired by Software Craftsmanship principles. Throughout the project, we found the following themes to be highly influential to the performed work:

- Maintain a long-term value vision, where both individuals and teams take accountability towards stakeholders, including other developers and business owners.
- Focus on short feedback loops at many levels, emphasising shortening feedback loops wherever possible while still providing clear, unambiguous, and relevant feedback.
- Use code kata exercises to show the expected product development methods, thus facilitating both short feedback loops and a long-term value vision.
- Foster a shared professional culture by keeping teams aligned and sharing a common way of working across development sites and time zones.

This article details how software craftsmanship works in practice. The project that we followed started with a single team of 9 developers but grew to encompass about 80 developers on two continents. The principles of clean code [87] and software craftsmanship were applied during the whole evolution of the system, and the consequences of adopting these principles are described in this article.

Studied product

We followed the evolution of a transaction-intensive application in the financial transfer domain from its conceptualization in the start-up phase, through its first installation at a customer site, to its expansion into approximately 15 different installations. It currently serves approximately 100 million customers. During our study, we also analyzed quantitative data from software development repositories and complemented data with interviews with individuals who were crucial to product development.

Approximately 25 developers contributed to the first release of the system in 2010. Development also involved a few requirement engineers, verification engineers, and various management roles. The number of developers fluctuated over time, with a mean of 48 developers and a maximum of 91. On average, each developer stayed nearly two years in the product, although five developers stayed the entire studied period, ending in 2016.

Study method

We conducted six interviews with developers who were involved in the project, including the lead system architect. To increase the reliability of our study, we also sought corroborating evidence from an archival analysis of several artifacts, including requirements, design documents, code repositories (Git), and fault management systems.

To gain a broader picture of software craftsmanship, we performed a systematic literature review using a process of forward and backward snowballing. We began with nine seed papers. After four snowball iterations, we found an additional nine papers on the subject. Based on these papers, we then constructed an informed and flexible interview protocol. After transcribing and coding the interviews, we also thematically coded eleven books referenced by the reviewed academic papers. Based on the findings from the papers, the books, and the interviews, we extracted several codes and established their relationships. We presented this information as a mind map. We published the complete details of these findings in [131].

From late 2009 until October 2016, the first author was part of a product development team. The second and third authors were used as support functions during the interviews and during the analysis phase of the study to counter any resulting bias.

5.2 Main observations

5.2.1 Individual accountability and team accountability

Being accountable is one of the main principles of software craftsmanship:

> It is an attitude of honesty, of honor, of self-respect, and of pride. It is a willingness to accept the dire responsibility of being a craftsman and an engineer. That responsibility includes working well and working clean. It includes communicating well and estimating faithfully. It involves managing your time and facing difficult risk-reward decisions. [88]

With mechanisms such as signing code artifacts, we ensure that individuals and teams feel accountable for the code they develop. Instilling accountability and pride are two solutions that can be used to counter the "just get it done" syndrome that seems to affect some knowledge-centric professions.

Indeed, accountability is now part of a professional practice that has been standardized, for example, in successful Open-Source projects, where a responsible release master signs off the code before being merged into the complete project. Since a team should be accountable and autonomous, this autonomy should enable them to take end-to-end responsibility for the developed features or components.

In the system that we studied, different ownership models were deployed to enable the level of autonomy described above. The main business logic was implemented in the context of a weak ownership model [44], where the teams took responsibility for the modules they developed, while simultaneously keeping an eye on changes that were made by developers outside their team. However, especially in the early stages of the product's development, the core of the transactions engine was subject to a much more formal code ownership model [44], where only a small set of developers were allowed to make changes in their closely guarded modules.

Main lessons — accountability

1. Development teams take on end-to-end responsibility to define, develop, and test solutions. Requirement engineers (e.g., a "Product Owner" role) performed initial conceptualization. Once development started, the development team constantly communicated

with the PO throughout the development work, seeking to clarify and get feedback on solutions and their consequences.

2. Code contributions are signed using personal certificates, thus emphasizing the importance of personal responsibility. The application has an internal Certificate Authority (CA) verifying the signatures.

3. Teams take part in the maintenance of the continuous integration environment. In particular, this work involves keeping the test base green, fixing flaky tests, and avoiding unnecessary long-running tests. When taking shortcuts such as disabling certain test cases to meet certain deadlines, specific tools are used to keep track of which developer that disabled what specific test case. After a grace period, these developers are reminded to take action for the test case in question.

4. Tests are as necessary as the production code itself and are used as "the continuously executing requirement specification of expected behavior" [88]. While most tests are automated, a small minority in the validation area is not automated (for example, validating instructions for end-users).

5. Automated regression tests grow faster than the production code, so the tests have to be layered, with testing done at the lowest layer where it makes sense. Because of the large amount of produced test code, it is equally important to clean up the test code and the production code.

5.2.2 Feedback loops

Shortening the Feedback Loop: Automation and Layering the architecture

Short feedback loops are a means to avoid bad habits and give developers early feedback on their work. Short feedback loops are probably the only way one can make progress with incremental, iterative development.

Using short feedback loops, organizations can adjust how the project should proceed before it progresses too far in an undesired direction.

Incremental development is the first stage at which feedback loops can be shortened. This approach can be conceived as a way of "growing software" instead of "building" it in a more traditional fashion. In parallel with the development process, prototyping and testing are obvious choices if one wishes to make the feedback loops shorter.

In the system that we studied, the organization strove to break the requirements down into smaller (*XSmall* and *Small* sizes) features, as can be seen in Table 5.1. Instead of spending months on developing several extensive features as chunks of related functions, the organization's focus was on obtaining early feedback, both from the QA teams, but even more importantly, from actual installations. Half of the *XSmall* stories were developed (including their analysis and design) in less than 22 days, the equivalent of one sprint. If we examine the pure development time (extracted from git-logs), we note that 50% of the stories were developed in less than 13 days. The days spent in QA portrays the time each feature spent in system testing and verification[2]. All of the interview participants mentioned short feedback loops as one of the organization's strong points.

However, to achieve the organization's goals, the system and software architecture need to be designed with testability in mind and support testing at different levels (such as the unit, integration, functional, and system levels) in a manner that is as simple as possible. The teams will also have to maintain a craftsman's attitude; *caring* about the code and *caring* about the test base.

Table 5.1: Elapsed Calendar Days Per Feature Size and Activity.

	Development		**No QA**		**QA Performed**	
Est.size	*#Stories*	$\hat{x}$	*#Stories*	%	*#Stories*	$\hat{x}$
X-Small	122	22	37	30.3%	85	7
Small	109	29	24	22.0%	85	8
Medium	72	47.5	10	13.9%	62	16.5
Large	13	62	1	7.7%	12	20.5

No QA is the number of features where planned system verification was deemed unnecessary.
$\hat{x}$ is the number of calendar days required to develop or system test 50% of the stories

[2]The development and testing departments used three-week sprints. The sprint time of the development and testing department might explain the average time that was spent on developing and testing large features (i.e., three sprints and one sprint, respectively)

Main lessons related to shortening feedback loops

1. Releases are frequently made, either to the system testing organization or directly to the customer. This requires automated test suites with regression tests in place, These tests need to be frequently executed and adequately maintained.

2. To enable frequent releases, we need to provide well-functioning and straightforward support for upgrades. This includes data models on persistent storage, in particular.

3. Operate a continuous integration loop, with teams constantly responding to feedback. As the product grows, the organization needs to optimize its feedback loops to keep them short. Continuous integration builds can be parallelized, and tests can be rewritten at a lower level.

4. Layered testing becomes crucial to shortening feedback loops. Each team should review and test what they develop, ensuring that the unit and functional tests they develop satisfy the new requirements. In some cases, this internal testing procedure (together with the regression test suite) can be sufficient to quality-assure the product.

5. Having short feedback loops and layered architectures enables developers to refactor their code, provided that they have a safety net to detect potential bugs introduced by the refactorings that were performed.

6. Human work is precious. Consequently, it should be focused on content, not style. Use tools that can provide automated feedback, such as static code analyzers and standard formatting tools, to allow colleagues to focus on reviewing the content instead of the style.

The dark side of test-focused development

When requirements are specified as executable test cases, conflicting forces emerge. Such forces need to be balanced against each other. For example, on

the one hand, each test case should verify as much functionality of the product as possible. On the other hand, for validation purposes, each test case should be readable by domain experts. Plain-text-based methods and tools such as BDD (Behavior-Driven Development) are often advocated since they enable the validation process [11], [58].

Because the number of possible test scenarios (including parameter validation and negative testing) usually outnumbers the product's actual functionality, the number of test cases would grow faster than the production code. However, plain-text languages and BDD tools often do not support refactoring and strongly typed navigation ("Find usages") out of the box.

Consequently, it is more difficult harder to refactor the test base into a more readable representation. Some organizations also have explicit policy rules against refactoring test code, citing the *"Quis custodiet ipsos custodes"* principle (i.e., raising the question: Who will ensure that the refactored test code behaves in the same way, finding the same faults as the original test code?)

Thus, there has to be an explicitly stated principle that tests need to be pushed down to the lowest level where they make sense. While keeping a single acceptance test case written in BDD style may well be preferable for validation purposes (i.e., ensuring that the function works as expected by the requirement owner), a caring developer would not use this type of testing to perform parameter validation of input parameters. Because of the large number of needed tests to perform adequate parameter validation, this type of testing is better done in a language that supports refactoring and automated restructuring. Typically, these are unit tests written in the same language as the production code. Performing parameter validation as unit tests also have the benefit of shortening the feedback loop. The architecture should thus encourage and enforce (to the extent possible) layered testing, and the organization should set itself the goal of performing tests at "the lowest level where they make sense".

The principles of (i) clean code and (ii) technical debt management apply to both production code and test code. In particular, care has to be taken when dealing with deprecated code to prevent the spread of its usage in the test base [129].

5.2.3 Skills

Pride vs. Humility

Developers are expected to show a degree of pride in their product and work process so that they are motivated to continue with their preferred ways of

working. Typically, this attitude will affect how the developer will respond to the inevitable time pressure. As one developer stated, related to the pressure to "deliver faster," without sufficient testing: "It's about what pride the team has. We don't hack together something and just leave it. When we are done, then we really *are* done."

Conversely, the developer's pride needs to be balanced against a sense of humility. Developers should realize that they are on a learning path, together with other stakeholders, exploring the potential solutions whose complexity needs to be weighed against the value of the problem it solves [60], [88].

Learning how to say Yes and No

Learning how to say "No" in a professional manner is an essential skill. For example, a developer should only commit to work tasks that the team estimates it can complete while upholding standards regarding, in particular, the verification and validation process. Usually, this involves time constraints and deadlines, which the organization tackled by: "Having a dialogue… 'No, we are not done yet, because… '"

Entering into a productive partnership requires establishing trust between individual teams and other stakeholders. Teams use common discussion forums to discuss acceptable solutions and acceptable criteria for the verification process. In our study, we noted that the lead developer/architect forum focused on and reviewed the solution. In contrast, the verification forum focused on what to test, how to test the solution efficiently and improve the verification procedure's overall performance.

When they were under time pressure, instead of performing fewer verification tests, the teams provided feedback to the stakeholders. Typically, this caused discussions to take place, with the intent to slice features into smaller parts, developing the most relevant (and therefore valuable) parts first. For a relationship of trust between development teams and stakeholders to be established, potential concerns have to be raised quickly so that the slicing is done as early as possible.

Developing individual skills and team skills

Individual skills are undoubtedly important for a craftsman, but equally important is the ability to develop and share these skills with others.

Relevant skills may relate to different areas, including:

- Tools, such as programming languages, IDEs, version control systems, and build tools.
- Work patterns for developing and verifying functions on the different test layers.
- The ability to conceptualize requirements (and communicate with other stakeholders) and turn those requirements into working tests and designs.
- Knowing where to go to find up-to-date requirements and how to raise requirement-related questions.

To build skills in its development teams, the organization used a set of structured exercises modeled according to the "code kata" concept [101]. By performing these exercises, the participant would be guided from an empty project into a fully-fledged web application (using the application and GUI framework used by the organization). As they were structured in a Test-Driven Development fashion, the katas emphasized how to test at the unit test level and the integration test level.

All interviewees appreciated the kata sessions, which allowed the group to learn about each other's strengths and weaknesses. As stated by a team member: "During the kata sessions, I realized that [in my newly formed team], we have different people with different backgrounds... I could see what mistake that they were doing and I could coach them."

5.2.4 Maintaining a shared professional culture

When upper management decided to outsource work, the lead developers, who had prior experience with outsourced products, understood the importance of maintaining a shared professional culture. To this aim, they required each onboarding team (consisting of 6-8 persons) to be present at the main site for training and during the development phase of their first feature. During this time (between 8 and 12 weeks for each team), each team would use the aforementioned kata exercises to learn about the application and the expected way of working. They would then develop their first complete product feature and establish professional connections with other people who possessed relevant knowledge. There is evidence that this contributed to establishing and maintaining a shared professional culture: "... work culture in [main site] and in India was almost similar... But in [other product] I see lots of difference between every corner of the world."

The organization used three different checklist-based *Definition of Done (DoD)* gates. The first of these DoD checklists focused on whether (or not) the feature was sufficiently conceptualized to make it clear what functional need that should be fulfilled. This checklist was signed-off by the requirement engineer responsible for the feature, before involving an entire development team of 6-8 people. The second DoD checklist focused on the actual development and functional verification of the feature in question. This checklist would be signed off by the team leader of the development team once the development and functional verification of the feature was complete. The third and final DoD checklist was signed off by the person responsible for system testing, who attested that either the feature had passed system verification and validation or that this had been deemed unnecessary. If this were the case, then the person responsible for system testing would provide reasons why this judgment was made.

Teams collaborated and helped each other find common synergies and solutions by discussing common problems and solutions in interest groups. These groups would participate in recurring meetings, with each team being represented. "It was not unusual to work across team boundaries... When we discussed and found that the structure would not hold any longer, we discussed how to set the new structure. Then two or three developers would do the restructuring and report back the progress."

5.2.5 The Diversity Aspect

Although there has been criticism that the term "craftsmanship" is gender-biased, we have chosen to keep the original term in this article since it is frequently used in the literature [83], [86]–[88], [91]. Other venues have taken different decisions, for instance, the SoCraTes[3] conference in Germany has now adopted a more gender-neutral name.

The actual Software Craftsmanship movement and the principles underlying Software Craftsmanship stress the importance of professionalism and inclusiveness regardless of a person's gender, culture or nationality.

Of the 155 developers who had contributed to the code base during the period studied, one-sixth (16.8%) were female, and 80.1% were male. For four individuals, gender information could not be deduced based on the name stored in the version control system. Two of our six interviewees were female.

[3]https://www.socrates-conference.de/history

5.3 Conclusions

In summary, there is little evidence of the "lone cowboy programmer" stereotype [13] in our model of Software Craftsmanship. Instead, we view Software Craftsmanship as "Agile done right", where Agile teams focus on the long-term value creation.

Our findings indicate a focus on individual and team *accountability*, where issues are raised as soon as they are identified. Trust is built between different stakeholders, and problems are managed directly, rather than by shifting blame to another person or team.

Similarly, we observed constant focus on *feedback loops* on many levels. Developers and managers are expected to shorten and streamline feedback loops to optimize the distribution and sharing of relevant information. Product development follows a highly iterative process and product managers frequently interact with developers, with the "real users" of the system, or with relevant proxies.

To enable frequent feedback, verification is highly automated using *regression tests*, which are developed *alongside the production code*, not "after the fact." Every development team takes responsibility for the regression test suite. We noted a culture where developers paid constant attention to the regression test suite. This included optimizing and stabilizing the tests to maintain a high level of confidence in them. Verification takes place on many levels, and the final validation takes place by using proxy customers, who act as the final users of the system.

Newcomers and onboarded teams are given time and relevant tutoring to become immersed in a *shared professional culture*. Cross-team forums are established to foster shared norms and behaviors. An open learning environment where expectations are made clear and relevant feedback is provided bolsters individual and team skills.

Finally, we found an environment where diversity is valued and where skills are utilized to provide maximum long-term value for the product as a whole.

References

[1] P. Abrahamsson and J. Koskela, “Extreme programming: A survey of empirical data from a controlled case study,” in *Proceedings - 2004 International Symposium on Empirical Software Engineering, ISESE ’04.*, IEEE, 2004, pp. 73–82. DOI: `10.1109/ISESE.2004.1334895`.

[2] E. Alégroth and J. Gonzalez-Huerta, “Towards a Mapping of Software Technical Debt onto Testware,” in *43rd Euromicro Conference on Software Engineering and Advanced Applications*, Vienna, Austria, 2017, pp. 404–411.

[3] M. I. Alhojailan, “Thematic Analysis: A Critical Review of Its Process and Evaluation,” *West East Journal of Social Sciences*, vol. 1, pp. 39–47, 2012.

[4] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, “Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162),” *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138, 2016, ISSN: 2192-5283. DOI: `10.4230/DagRep.6.4.110`.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture,” *IEEE Software*, vol. 33, no. 3, pp. 42–52, May 2016, ISSN: 07407459. DOI: `10.1109/MS.2016.64`.

[6] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Upper Saddle River, NJ, USA: Pearson Education, Inc., 2013, ISBN: 978-0321815736.

[7] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Professional, 1999, ISBN: 978-0201616415.

[8] ——, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0321146530.

[9] M. Beedle, M. Devos, Y. Sharon, K. Schwaber, and J. Sutherland, "Scrum: An extension pattern language for hyperproductive software development," in *Pattern languages of program design, 4*, N. Harrison, B. Foote, and H. Rohnert, Eds., vol. 4, Reading, MA: Addison Wesley, 2000, ch. 28, pp. 637–651, ISBN: 978-0201433043.

[10] I. Bergström and A. F. Blackwell, "The practices of programming," in *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, vol. 2016-Novem, 2016, pp. 190–198, ISBN: 978-1509002528. DOI: 10.1109/VLHCC.2016.7739684.

[11] E. Bjarnason, M. Unterkalmsteiner, M. Borg, and E. Engström, "A multi-case study of agile requirements engineering and the use of test cases as requirements," *Information and Software Technology*, vol. 77, pp. 61–79, 2016.

[12] S. Blakstad and R. Allen, *FinTech Revolution: Universal Inclusion in the New Financial Ecosystem.* Springer International Publishing, 2018, ISBN: 978-3319760148.

[13] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: ACM, 2006, pp. 12–29, ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134288. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134288.

[14] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, Jan. 2006, ISSN: 1478-0887. DOI: 10.1191/1478088706qp063oa. arXiv: 1011.1669.

[15] ——, "What can "thematic analysis" offer health and wellbeing researchers?" *International Journal of Qualitative Studies on Health and Well-being*, vol. 9, pp. 20–22, 2014, ISSN: 17482631. DOI: 10.3402/qhw.v9.26152.

[16] R. Britto, D. Šmite, and L.-O. Damm, "Software Architects in Large-Scale Distributed Projects: An Ericsson Case Study," *IEEE Software*, vol. 33, no. 6, pp. 48–55, Nov. 2016, ISSN: 0740-7459. DOI: 10.1109/MS.2016.146. [Online]. Available: http://ieeexplore.ieee.org/document/7725230/.

[17] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Pearson Education, 1995, ISBN: 978-0132119160.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture. A System of Patterns.* Chichester: John Wiley & Sons, 1996, vol. Vol. 1, p. 476, ISBN: 0471958697.

[19] C. Calhoun, *Critical Social Theory: Culture, History, and the Challenge of Difference.* Oxford, UK: Wiley-Blackwell, 1995, ISBN: 978-1557862884.

[20] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors limiting industrial adoption of test driven development: A systematic review," in *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011*, Berlin, Germany, 2011, pp. 337–346, ISBN: 978-0769543420.

[21] O. Cawley, X. Wang, and I. Richardson, "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, 2010, pp. 31–36, ISBN: 3642164153. DOI: `10.1007/978-3-642-16416-3_4`.

[22] P. Chatzipetrou, D. Šmite, and R. van Solingen, "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*, ser. ESEM '18, Oulu, Finland: Association for Computing Machinery, 2018, ISBN: 978-1450358231. DOI: `10.1145/3239235.3267431`. [Online]. Available: `https://doi.org/10.1145/3239235.3267431`.

[23] P. C. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Meson, R. Nord, J. Stafford, P. Merson, R. Nord, and J. Stafford, *Documenting software architectures: views and beyond*, 2nd Editio. Pearson Education, 2010. [Online]. Available: `http://dl.acm.org/citation.cfm?id=599933`.

[24] Z. Codabux and B. Williams, "Managing Technical Debt: An Industrial Case Study," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, 2013, pp. 8–15, ISBN: 978-1467364430.

[25] M. Cohn, *Succeeding with Agile: Software Development Using Scrum.* Pearson Education, 2009, p. 504, ISBN: 0321660560.

[26] CollabNet VersionOne, "The 13th annual STATE OF AGILE Report - 2018," Tech. Rep., 2019. [Online]. Available: `https://stateofagile.com/#ufh-i-613553418-13th-annual-state-of-agile-report/7027494`.

[27] J. O. Coplien, "Borland software craftsmanship: A new look at process, quality and productivity," in *Proceedings of the 5th Annual Borland International Conference*, Orlando, FL, USA, 1994.

[28] L. Crispin and J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, ser. Addison-Wesley Signature Series (Cohn). Pearson Education, 2008, ISBN: 978-0321616937.

[29] D. S. Cruzes and T. Dybå, "Recommended Steps for Thematic Synthesis in Software Engineering," *2011 International Symposium on Empirical Software Engineering and Measurement*, no. 7491, pp. 275–284, 2011. DOI: 10.1109/esem.2011.36.

[30] D. S. Cruzes and T. Dybå, "Research synthesis in software engineering: A tertiary study," *Information and Software Technology*, vol. 53, no. 5, pp. 440–455, 2011, ISSN: 09505849. DOI: 10.1016/j.infsof.2011.01.004. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2011.01.004.

[31] D. S. Cruzes, T. Dybå, P. Runeson, and M. Höst, "Case studies synthesis: a thematic, cross-case, and narrative synthesis worked example," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1634–1665, 2015, ISSN: 15737616. DOI: 10.1007/s10664-014-9326-8.

[32] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[33] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Commun. ACM*, vol. 31, no. 11, pp. 1268–1287, Nov. 1988, ISSN: 0001-0782. DOI: 10.1145/50087.50089. [Online]. Available: https://doi.org/10.1145/50087.50089.

[34] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," *IEEE Software*, vol. 29, no. 6, pp. 34–42, 2012.

[35] P. Diebold and M. Dahlem, "Agile practices in practice - A mapping study," in *18th International Conference on Evaluation and Assessment in Software Engineering*, Association for Computing Machinery, 2014, ISBN: 978-1450324762. DOI: 10.1145/2601248.2601254.

[36] E. W. Dijkstra, "On the role of scientific thought," in *Selected writings on computing: a personal perspective*, Springer, 1982, pp. 60–66.

[37] E. W. Dijkstra, *On the reliability of programs.* [Online]. Available: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html (visited on 08/19/2021).

[38] T. Dogša and D. Batič, "The effectiveness of test-driven development: An industrial case study," *Software Quality Journal*, vol. 19, no. 4, pp. 643–661, 2011, ISSN: 15731367. DOI: 10.1007/s11219-011-9130-2.

[39] S. Easterbrook, J. Singer, M. A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. Sjøberg, Eds., 2008, pp. 285–311, ISBN: 978-1848000438. DOI: 10.1007/978-1-84800-044-5_11.

[40] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, Mar. 2005, ISSN: 0098-5589.

[41] M. Feyh and K. Petersen, "Lean software development measures and indicators - A systematic mapping study," in *Lecture Notes in Business Information Processing*, vol. 167, Springer Verlag, 2013, pp. 32–47, ISBN: 978-3642449291. DOI: 10.1007/978-3-642-44930-7_3.

[42] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative Inquiry*, vol. 12, no. 2, pp. 219–245, 2006. DOI: 10.1177/1077800405284363. [Online]. Available: https://doi.org/10.1177/1077800405284363.

[43] M. Fowler, *BeckDesignRules*, 2015. [Online]. Available: https://martinfowler.com/articles/BeckDesignRules.html (visited on 08/08/2020).

[44] ——, *Code Ownership*, 2006. [Online]. Available: https://martinfowler.com/bliki/CodeOwnership.html (visited on 06/18/2021).

[45] ——, *Microservices*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 01/08/2020).

[46] M. Fowler, K. Beck, J. Brant, and W. Opdyke, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999, ISBN: 978-0201485677.

[47] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A dissection of the test-driven development process: Does it really matter to test-first or to test-last?" *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 597–614, 2017. DOI: 10.1109/TSE.2016.2616877.

[48] D. Fucci, B. Turhan, and M. Oivo, “On the effects of programming and testing skills on external quality and productivity in a test-driven development context,” in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering - EASE ’15*, 2015, pp. 1–6, ISBN: 978-1450333504. DOI: 10.1145/2745802.2745826. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2745802.2745826.

[49] K. Gai, M. Qiu, and X. Sun, “A survey on fintech,” *Journal of Network and Computer Applications*, vol. 103, pp. 262–273, 2018, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2017.10.011.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Pearson Education, 1994, ISBN: 978-0321700698.

[51] V. Garousi, M. Felderer, and M. V. Mäntylä, “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering,” *Information and Software Technology*, vol. 106, no. May 2018, pp. 101–121, 2019, ISSN: 09505849. DOI: 10.1016/j.infsof.2018.09.006. arXiv: 1707.02553. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.09.006.

[52] T. Gilbert, *Human Competence: Engineering Worthy Performance*. McGraw-Hill, 1978, ISBN: 978-0070232174.

[53] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967.

[54] D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson, “Consequences of unhappiness while developing software,” *Proceedings - 2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering, SEmotion 2017*, no. SEmotion, pp. 42–47, 2017. DOI: 10.1109/SEmotion.2017.5. arXiv: 1701.05789.

[55] D. Graziotin, X. Wang, and P. Abrahamsson, “Are Happy Developers More Productive?” In *Product-Focused Software Process Improvement*, J. Heidrich, M. Oivo, A. Jedlitschka, and M. T. Baldassarre, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 50–64, ISBN: 978-3-642-39259-7.

[56] C. Haines, *Understanding the 4 Rules of Simple Design*. Leanpub, 2014. [Online]. Available: https://leanpub.com/4rulesofsimpledesign (visited on 04/19/2018).

[57] B. Haugset and G. K. Hanssen, "Automated acceptance testing: A literature review and an industrial case study," in *Agile 2008 Conference*, Toronto, Canada, 2008, pp. 27–38.

[58] B. Haugset and T. Stalhane, "Automated acceptance testing as an agile requirements engineering practice," in *2012 45th Hawaii International Conference on System Sciences*, 2012, pp. 5289–5298. DOI: `10.1109/HICSS.2012.127`.

[59] R. Hoda, N. Salleh, J. Grundy, and H. M. Tee, "Systematic literature reviews in agile software development: A tertiary study," *Information and Software Technology*, vol. 85, pp. 60–70, 2017, ISSN: 09505849. DOI: `10.1016/j.infsof.2017.01.007`. [Online]. Available: `http://dx.doi.org/10.1016/j.infsof.2017.01.007`.

[60] D. Hoover and A. Oshineye, *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman*, ser. Theory in practice. O'Reilly Media, 2009, ISBN: 978-1449379407.

[61] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999, ISBN: 978-0132119177.

[62] M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," *Empirical Software Engineering*, vol. 16, no. 3, pp. 365–395, 2011.

[63] I. Jacobson and E. Seidewitz, "A new software engineering," *Queue*, vol. 12, no. 10, 30:30–30:38, Oct. 2014, ISSN: 1542-7730. DOI: `10.1145/2685690.2693160`. [Online]. Available: `http://doi.acm.org/10.1145/2685690.2693160`.

[64] S. Jalali and C. Wohlin, "Global software engineering and agile practices: A systematic review," *Journal of software: Evolution and Process*, vol. 24, no. 6, pp. 643–659, 2012.

[65] D. Karlström and P. Runeson, "Combining Agile Methods with Stage-Gate Project Management," *IEEE Software*, no. May/June, pp. 43–49, 2005.

[66] G. L. Kelling, J. Q. Wilson, *et al.*, "Broken windows," *Atlantic monthly*, vol. 249, no. 3, pp. 29–38, 1982.

[67] M. Kim, T. Zimmermann, and N. Nagappan, “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*, New York, New York, USA: ACM Press, 2012, p. 1, ISBN: 978-1450316149. DOI: 10.1145/2393596.2393655. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2393596.2393655.

[68] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, “Systematic literature reviews in software engineering - A systematic literature review,” *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009, ISSN: 09505849. DOI: 10.1016/j.infsof.2008.09.009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.09.009.

[69] B. A. Kitchenham, T. Dybå, and M. Jørgensen, “Evidence-based Software Engineering,” in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 273–281. DOI: 10.1109/ICSE.2004.1317449.

[70] H. K. Klein and M. D. Myers, “A set of principles for conducting and evaluating interpretive field studies in information systems,” *MIS Quarterly*, vol. 23, no. 1, pp. 67–93, 1999. DOI: 10.2307/249410. [Online]. Available: http://www.jstor.org/stable/249410.

[71] D. E. Knuth, “Literate programming,” *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[72] P. Kruchten, R. L. Nord, and I. Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development.* Addison-Wesley Professional, 2019, ISBN: 978-0135645932.

[73] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018, ISSN: 15737616. DOI: 10.1007/s10664-017-9521-5. arXiv: 1709.04621.

[74] E. Kupiainen, M. V. Mäntylä, and J. Itkonen, *Using metrics in Agile and Lean software development - A systematic literature review of industrial studies*, 2015. DOI: 10.1016/j.infsof.2015.02.005.

[75] C. Larman and B. Vodde, *Scaling Lean and Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*, ser. Agile Software Development Series. Pearson Education, 2008, ISBN: 978-0321617149.

[76] I. Lee and Y. J. Shin, “Fintech: Ecosystem, business models, investment decisions, and challenges,” *Business Horizons*, vol. 61, no. 1, pp. 35–46, 2018, ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2017.09.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0007681317301246.

[77] P. Lenberg, R. Feldt, and L. G. Wallgren, “Behavioral software engineering: A definition and systematic literature review,” *Journal of Systems and Software*, vol. 107, pp. 15–37, 2015, ISSN: 01641212. DOI: 10.1016/j.jss.2015.04.084.

[78] K. Lewin, “Action research and minority problems,” *Journal of Social Issues*, vol. 2, no. 4, pp. 34–46, 1946.

[79] R. Lindell, “Crafting interaction: The epistemology of modern programming,” *Personal Ubiquitous Comput.*, vol. 18, no. 3, pp. 613–624, Mar. 2014, ISSN: 1617-4909. DOI: 10.1007/s00779-013-0687-6. [Online]. Available: http://dx.doi.org/10.1007/s00779-013-0687-6.

[80] ——, “The Craft of Programming Interaction,” in *Proceedings of International Workshop on the Interplay between User Experience Evaluation and Software Development (I-UxSED 2012)*, 2012, pp. 26–30.

[81] J. Lingel and T. Regan, “"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*, ser. CSCW ’14, Baltimore, Maryland, USA: ACM, 2014, pp. 295–304, ISBN: 978-1-4503-2540-0. DOI: 10.1145/2531602.2531614. [Online]. Available: http://doi.acm.org/10.1145/2531602.2531614.

[82] P. Lucena and L. P. Tizzei, “Applying Software Craftsmanship Practices to a Scrum Project: an Experience Report,” in *I Workshop sobre Aspectos Sociais, Humanos e Econômicos de Software (WASHES 2016)*, Maceió, Alagoas, Brazil, 2016. arXiv: 1611.05789. [Online]. Available: http://arxiv.org/abs/1611.05789.

[83] S. Mancuso, *The Software Craftsman: Professionalism, Pragmatism, Pride.* Pearson Education, 2014, ISBN: 978-0134052588.

[84] G. Marcionetti, F. Cannizzo, and P. Moser, “The toolbox of a successful software craftsman,” in *Engineering of Computer-Based Systems, IEEE International Conference on the(ECBS)*, vol. 00, Mar. 2008, pp. 389–397. DOI: 10.1109/ECBS.2008.48. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ECBS.2008.48.

[85] R. C. Martin, *Clean Agile: Back to Basics*. Pearson Education, 2020, ISBN: 978-0-13-578186-9.

[86] ——, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson Education, 2017, ISBN: 978-0134494326.

[87] ——, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, ISBN: 978-0132350884.

[88] ——, *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall, 2011, ISBN: 978-0137081073.

[89] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *25th International Conference on Software Engineering*, vol. 6, Portland, OR USA, 2003, pp. 564–569, ISBN: 0-7695-1877-X.

[90] J. A. Maxwell, *Qualitative Research Design: An Interactive Approach*. Thousand Oaks, CA, US: Sage Publishing, Inc., 1996, ISBN: 978-1412981194.

[91] P. McBreen, *Software Craftsmanship: The New Imperative*. Addison-Wesley, 2002, ISBN: 978-0201733860.

[92] J. McCarthy, *Dynamics of software development*. Microsoft Press, Redmond, WA, 1995, vol. 3, ISBN: 978-1556158230.

[93] G. I. Melnik, "Empirical analyses of executable acceptance test driven development," Ph.D. dissertation, University of Calgary, Calgary, Canada, 2007, ISBN: 978-0-494-33806-3.

[94] L. Menand, Ed., *Pragmatism: A Reader*. New York, NY, USA: Random House, Inc., 1997, ISBN: 978-0679775447.

[95] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007, ISBN: 978-0131495050.

[96] E. Mourão, J. F. Pimentel, L. Murta, M. Kalinowski, E. Mendes, and C. Wohlin, "On the performance of hybrid search strategies for systematic literature reviews in software engineering," *Information and Software Technology*, vol. 123, p. 106 294, 2020, ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2020.106294`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0950584920300446`.

[97] H. Munir, M. Moayyed, and K. Petersen, "Considering rigor and relevance when evaluating test driven development: A systematic review," 2014. DOI: `10.1016/j.infsof.2014.01.002`. [Online]. Available: `http://dx.doi.org/10.1016/j.infsof.2014.01.002`.

[98] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: Results and experiences of four industrial teams," *Empirical Software Engineering*, vol. 13, no. 3, pp. 289–302, 2008, ISSN: 13823256.

[99] M. Paasivaara and C. Lassenius, "Communities of practice in a large distributed agile software development organization – case ericsson," *Information and Software Technology*, vol. 56, no. 12, pp. 1556–1577, 2014, Special issue: Human Factors in Software Development, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.06.008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584914001475.

[100] M. Pančur and M. Ciglarič, "Impact of test-driven development on productivity, code and tests: A controlled experiment," *Information and Software Technology*, vol. 53, no. 6, pp. 557–573, 2011, ISSN: 09505849. DOI: 10.1016/j.infsof.2011.02.002.

[101] D. Parsons, A. Mathrani, T. Susnjak, and A. Leist, "Coderetreats: Reflective practice and the game of life," *IEEE Software*, vol. 31, no. 4, pp. 58–64, Jul. 2014, ISSN: 0740-7459. DOI: 10.1109/MS.2014.25.

[102] D. Parsons, T. Susnjak, and A. Mathrani, "Design from detail: Analyzing data from a global day of coderetreat," *Information and Software Technology*, vol. 75, pp. 39–55, 2016, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.03.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300519.

[103] K. Petersen, "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, 2012, pp. 108–122, ISBN: 978-3642303494. DOI: 10.1007/978-3-642-30350-0_8.

[104] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattson, "Systematic Mapping Studies in Software Engineering," in *Evaluation and Assessment in Software Engineering (EASE)*, vol. 12, 2008.

[105] K. Petersen and C. Wohlin, "Measuring the flow in lean software development," *Software - Practice and Experience*, vol. 41, no. 9, pp. 975–996, 2011, ISSN: 00380644. DOI: 10.1002/spe.975.

[106] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 2003, ISBN: 978-0321150783.

[107] K. Pugh, *Lean-agile acceptance test driven development : better software through collaboration*. Addison-Wesley, 2010, ISBN: 978-0321714084.

[108] B. Pyritz, "Craftsmanship versus engineering: Computer programming - an art or a science?" *Bell Labs Technical Journal*, vol. 8, no. 3, pp. 101–104, 2003. DOI: 10.1002/bltj.10079. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/bltj.10079.

[109] P. Ralph and E. Tempero, "Construct validity in software engineering research and software metrics," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, ser. EASE'18, Christchurch, New Zealand: Association for Computing Machinery, 2018, pp. 13–23, ISBN: 978-1450364034. DOI: 10.1145/3210459.3210461. [Online]. Available: https://doi-org.miman.bib.bth.se/10.1145/3210459.3210461.

[110] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., 2015, ISBN: 978-1491924242.

[111] C. Robson and K. McCartan, *Real world research*, 4th Ed. John Wiley & Sons, 2016, ISBN: 978-1-118-74523-6.

[112] P. Rodríguez, K. Mikkonen, P. Kuvaja, M. Oivo, and J. Garbajosa, "Building lean thinking in a telecom software development organization: Strengths and challenges," in *Proceedings of the 2013 International Conference on Software and System Process*, ser. ICSSP 2013, San Francisco, CA, USA: ACM, 2013, pp. 98–107, ISBN: 978-1-4503-2062-7. DOI: 10.1145/2486046.2486064. [Online]. Available: http://doi.acm.org/10.1145/2486046.2486064.

[113] W. W. Royce, "Managing the development of large software systems," in *Proceedings, IEEE WESCON*, 1970, pp. 1–9. DOI: 10.1016/0378-4754(91)90107-E.

[114] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012, ISBN: 978-1-118-10435-4.

[115] D. Salah, R. F. Paige, and P. Cairns, "A systematic literature review for Agile development processes and user centred design integration," in *18th International Conference on Evaluation and Assessment in Software Engineering*, London, UK: Association for Computing Machinery, 2014, ISBN: 978-1450324762. DOI: 10.1145/2601248.2601276.

[116] J. Saldana, *Coding Manual for Qualitative Researchers*, 3rd. Sage Publications, 2015, p. 223, ISBN: 1473902495.

[117] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs," *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, pp. 400–410, 2017. DOI: 10.1109/ICSME.2016.64.

[118] ——, "To react, or not to react: Patterns of reaction to API deprecation," *Empirical Software Engineering*, pp. 3824–3870, 2019, ISSN: 15737616. DOI: 10.1007/s10664-019-09713-w.

[119] T. Sedano, "Towards teaching software craftsmanship," in *2012 IEEE 25th Conference on Software Engineering Education and Training*, Apr. 2012, pp. 95–99. DOI: 10.1109/CSEET.2012.29.

[120] P. Seibel, *Coders at Work: Reflections on the Craft of Programming*, ser. IT Pro. Apress, 2009, ISBN: 978-1430219491.

[121] R. Sennett, *The Craftsman*. Yale University Press, 2008, ISBN: 978-0300149555.

[122] H. Sharp, Y. Dittrich, and C. R. B. de Souza, "The role of ethnographic studies in empirical software engineering," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 786–804, 2016. DOI: 10.1109/TSE.2016.2519887.

[123] A. Silva, T. Araújo, J. Nunes, M. Perkusich, E. Dilorenzo, H. Almeida, and A. Perkusich, "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*, ser. EASE'17, Karlskrona, Sweden: Association for Computing Machinery, 2017, pp. 364–373, ISBN: 978-1450348041. DOI: 10.1145/3084226.3084262. [Online]. Available: https://doi.org/10.1145/3084226.3084262.

[124] D. Šmite, N. B. Moe, G. Levinta, and M. Floryan, "Spotify guilds: How to succeed with knowledge sharing in large-scale agile organizations," *IEEE Software*, vol. 36, no. 2, pp. 51–57, 2019. DOI: 10.1109/MS.2018.2886178.

[125] D. Šmite, N. B. Moe, M. Floryan, G. Levinta, and P. Chatzipetrou, "Spotify guilds," *Commun. ACM*, vol. 63, no. 3, pp. 56–61, Feb. 2020, ISSN: 0001-0782. DOI: 10.1145/3343146. [Online]. Available: https://doi.org/10.1145/3343146.

[126] W. Snipes and S. Ramaswamy, "A proposed sizing model for managing 3rd party code technical debt," *Proceedings - International Conference on Software Engineering*, pp. 72–75, 2018, ISSN: 02705257. DOI: 10.1145/3194164.3194179.

[127] K. J. Stol, P. Ralph, and B. Fitzgerald, "Grounded Theory in Software Engineering Research: A Critical Review and Guidelines," in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2016, pp. 120–131, ISBN: 978-1450339001. DOI: 10.1145/2884781.2884833.

[128] A. Sundelin, J. Gonzalez-Huerta, and K. Wnuk, "Test-Driving FinTech Product Development: An Experience Report," in *International Conference on Product-Focused Software Process Improvement*, Springer, 2018, pp. 219–226, ISBN: 978-3030036737. DOI: 10.1007/978-3-030-03673-7_16.

[129] ——, "The hidden cost of backward compatibility: When deprecation turns into technical debt - An experience report," in *Proceedings - 2020 IEEE/ACM International Conference on Technical Debt, TechDebt 2020*, 2020, ISBN: 978-1450379601. DOI: 10.1145/3387906.3388629.

[130] A. Sundelin, J. Gonzalez-Huerta, K. Wnuk, and T. Gorschek, "Dear Lone Cowboy Programmer - your days are numbered!" *Communications of the ACM*, Submitted 2021-07-22.

[131] ——, "Towards an anatomy of software craftsmanship," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, to appear, 2021. DOI: 10.1145/3468504.

[132] P. Taylor, "Vernacularism in software design practice: Does craftsmanship have a place in software engineering?" *Australasian Journal of Information Systems*, vol. 11, no. 1, 2003, ISSN: 1449-8618. DOI: 10.3127/ajis.v11i1.143. [Online]. Available: https://journal.acs.org.au/index.php/ajis/article/view/143.

[133] D. Thomas, "Professional developers practice their kata to stay sharp.," *Journal of Object Technology*, vol. 9, pp. 23–25, Mar. 2010. DOI: 10.5381/jot.2010.9.2.c3.

[134] A. Tosun, O. Dieste, D. Fucci, S. Vegas, B. Turhan, H. Erdogmus, A. Santos, M. Oivo, K. Toro, J. Jarvinen, and N. Juristo, "An industry experiment on the effects of test-driven development on external quality and productivity," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2763–2805, Dec. 2017, ISSN: 15737616. DOI: 10.1007/s10664-016-9490-0.

[135] R. Vallon, B. J. da Silva Estácio, R. Prikladnicki, and T. Grechenig, "Systematic literature review on agile practices in global software development," *Information and Software Technology*, vol. 96, no. April 2017, pp. 161–180, 2018, ISSN: 09505849. DOI: 10.1016/j.infsof.2017.12.004. [Online]. Available: https://doi.org/10.1016/j.infsof.2017.12.004.

[136] E. Wenger, *Communities of practice: Learning, meaning, and identity.* 1999, ISBN: 978-0521663632.

[137] R. Westrum, "A typology of organisational cultures," *BMJ Quality & Safety*, vol. 13, no. suppl 2, pp. ii22–ii27, 2004.

[138] R. Winter, *Agile Performance Improvement: The New Synergy of Agile and Human Performance Technology.* Apress, 2015, ISBN: 978-1484208922.

[139] N. Wirth, "A Brief History of Software Engineering," *IEEE Annals of the History of Computing*, vol. 30, no. 3, pp. 32–39, 2008, ISSN: 10586180. DOI: 10.1109/MAHC.2008.33.

[140] C. Wohlin, "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*, ser. EASE '14, London, England, United Kingdom: ACM, 2014, 38:1–38:10, ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601268. [Online]. Available: http://doi.acm.org/10.1145/2601248.2601268.

[141] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering.* Berlin: Springer, 2012, ISBN: 978-3-642-29043-5.

[142] R. K. Yin, *Case Study Research: Design and Methods*, 5th Ed. Sage Publications, Inc., 2014, ISBN: 978-1-4522-4256-9.

[143] E. Zabardast, J. Gonzalez-Huerta, and D. Šmite, "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, Aug. 2020, pp. 376–384, ISBN: 978-1728195322. DOI: 10.1109/SEAA51224.2020.00068. [Online]. Available: https://ieeexplore.ieee.org/document/9226289/.

[144] O. Zimmermann, "Microservices tenets: Agile approach to service development and deployment," *Computer Science - Research and Development*, vol. 32, no. 3-4, pp. 301–310, 2017, ISSN: 18652042. DOI: 10.1007/s00450-016-0337-0.