Dear Lone Cowboy Programmer - your days are numbered!

Anders Sundelin*
Ericsson AB, and
Department of Software Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
anders.sundelin@ericsson.com

Javier Gonzalez-Huerta
Department of Software Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
jgh@bth.se

Krzysztof Wnuk
Department of Software Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
krw@bth.se

Tony Gorschek
Department of Software Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
tony.gorschek@bth.se

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.

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 [3]. 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.
- Not only responding to change, but also steadily adding value.
- Not only individuals and interactions, but also a community of professionals.

*industrial PhD Candidate at BTH

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

- Not only customer collaboration, but also productive partnerships.

Several scholars have commented on software craftsmanship. For example, Jacobson [7] has argued that engineering is a craft that is supported by theory, while Bergtröm and Blackwell [1] argue that professional practice is "craftwork." In their empirical study, Lingel and Regan [8] 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 [9], 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 [11] and software craftsmanship were applied during the whole evolution of the system, and the consequences of adopting these principles are described in this article.

The product that we studied
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.

How we performed the study
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 [17].
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.

2. MAIN OBSERVATIONS

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. [12]

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 [4], 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 [4], 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" [12]. 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.

2.2 Feedback loops

2.2.1 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 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 1: Elapsed calendar days per feature size (using T-Shirt sizes estimated before starting development) and activity. No QA is the number of features where planned system verification was deemed unnecessary, $\hat{x}$ is the number of days required to develop or system test 50% of the stories

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

[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.

2.2.2 *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 [2, 5].

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 [16].

2.3 Skills

2.3.1 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 [6, 12].

2.3.2 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.

2.3.3 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 [15]. 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."

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."

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 [10, 11, 12, 13, 14]. 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. CONCLUSIONS

In summary, there is little evidence of the "lone cowboy programmer" stereotype [3] 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.

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

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.

4. ACKNOWLEDGMENTS

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

5. REFERENCES

[1] I. Bergström and A. F. Blackwell. The practices of programming. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, volume 2016-Novem, pages 190–198, 2016.

[2] 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*, 77:61–79, 2016.

[3] B. Boehm. A View of 20th and 21st Century Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 12–29, New York, NY, USA, 2006. ACM.

[4] M. Fowler. Code Ownership. At `https://martinfowler.com/bliki/CodeOwnership.html`, seen: 2021-06-18.

[5] B. Haugset and T. Stalhane. Automated acceptance testing as an agile requirements engineering practice. In *2012 45th Hawaii International Conference on System Sciences*, pages 5289–5298, 2012.

[6] D. Hoover and A. Oshineye. *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman.* Theory in practice. O'Reilly Media, 2009.

[7] I. Jacobson and E. Seidewitz. A new software engineering. *Queue*, 12(10):30:30–30:38, Oct. 2014.

[8] 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, CSCW '14, pages 295–304, New York, NY, USA, 2014. ACM.

[9] P. Lucena and L. P. Tizzei. Applying software craftsmanship practices to a scrum project: an experience report. *CoRR*, abs/1611.05789, 2016.

[10] S. Mancuso. *The Software Craftsman: Professionalism, Pragmatism, Pride*. Robert C. Martin Series. Pearson Education, 2014.

[11] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008.

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

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

[14] P. McBreen. *Software Craftsmanship: The New Imperative*. Addison-Wesley, 2002.

[15] D. Parsons, A. Mathrani, T. Susnjak, and A. Leist. Coderetreats: Reflective practice and the game of life. *IEEE Software*, 31(4):58–64, July 2014.

[16] 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.

[17] A. Sundelin, J. Gonzalez-Huerta, K. Wnuk, and T. Gorschek. Towards an anatomy of software craftsmanship. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, page to appear, 2021.