# Agile Practices in Regulated Railway Software Development

Henrik Jonsson
System Development Department
Etteplan Industry AB
Västerås, Sweden
henrik.jonsson@etteplan.com

Stig Larsson and Sasikumar Punnekkat
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
{stig.larsson,sasikumar.punnekkat}@mdh.se

**Abstract—Complex software is becoming an important component of modern safety-critical systems. To assure the correct function of such software, the development processes are heavily regulated by international standards, often making the process very rigid, unable to accommodate changes, causing late integration and increasing the cost of development. Agile methods have been introduced to address these issues in several software domains, but their use in safety-critical applications remains to be investigated.**

**This paper provides an initial analysis of agile practices in the context of software development for the European railway sector, regulated by the EN 50128 standard. The study complements previous studies on the use of agile methods in other regulated domains.**

**A systematic mapping between EN 50128 requirements and agile practices showed that all practices support some objectives of the standard. Important supporting features recognized were focus on simple design, test automation, coding standards, continuous integration and validation. However, several problematic areas were also identified, including vague requirement analysis and change management. Most agile practices must be adapted to suit regulated software development and this analysis outlines a subset of the required changes.**

***Keywords-software engineering, software development processes, agile practices, safety-critical systems, railway, EN 50128***

## I. INTRODUCTION

While agile methods have widely been adopted in many domains of software industry, their use in regulated and safety-critical domains is still limited. In the latter, use of document-heavy plan-driven processes (such as “waterfall”) is still common, and more or less assumed by standards on functional safety. However, the ever-increasing usage and complexity of software in safety-critical systems, calls for more efficient and flexible ways to produce the software. Today, the cost and time to produce safety software is perceived to be too high, especially in domains where the combination of regulations and software implementation of safety related functions are relatively new, for instance in the railway sector.

Current standards on functional safety often describe software development as a strict sequential process with distinct phases for requirements, architecture, design and component coding, and corresponding testing at increasing levels in the end. Such models are also referred to as plan-driven, since they emphasize detailed planning and specification before proceeding to implementation and final testing. Problems with strict sequential plan-driven methods include

- Large and late integration of system. Errors found in these late stages will be costly to fix and cause delays of the release.
- Little involvement of the customer and end-user increases the risk of producing unsatisfactory solutions.
- Difficulty to articulate and specify all requirements in the beginning of the project.
- Difficulty to address new requirements and findings during the life cycle
- Requires many documents or other artifacts, which are costly to produce and maintain, and are often awkwardly produced by software engineers.

Much as a reaction to plan-driven and document-centric methods, so called agile methods have been developed and used [12]. Agile methods addresses the issues above mainly by working closer to the customer, focusing on team collaboration, and by specifying, designing, building, integrating, testing and validating the system iteratively and incrementally. The goal in agile development is that each small release should provide business value in the form of fully working software. Besides the value of being able to sell software or related services, agile methods recognized that there is also a value to be able to demonstrate and validate the software functionality early, even though not all of the functionality is included. This gives important feedback early, as well as an opportunity to change direction according to new technical or business conditions. In contrast to other iterative methods, such as the Rational Unified Process [11], agile methods emphasize non-written communication and close team and customer collaboration. In agile methods planning is still central, but is only made detailed on short-term. Agile work can still be very socially and technically disciplined, but often proposes different practices than plan-driven methods.

Traditionally safety regulation has been most developed in the nuclear, avionics and the medical sectors. Most research on regulated software development environment, now also extended to agile, has also focused on these areas. However, as the general public safety-awareness rises and the use of software increases, sectors like automotive and railway need more regulation and research attention.

The contribution of this paper is to analyze the agile practices of eXtreme Programming (XP) [1] against the requirements of the railway standard EN 50128 [2]. The objective was to identify any requirement or aspect of the standard in conflict with, or supported by, agile practices. Furthermore, propositions on how to adopt the practices in this regulated environment are provided.

EN 50128 is a European standard that regulates development, deployment and maintenance of safety-related software intended for railway applications. It contains requirements on the developing organization (roles and competences), life-cycle (phases, documentation and methods) and software assurance (testing, verification, validation and quality assurance and assessment).

The rest of the paper is organized as follows: In section II related publications on agile methods within the regulated domains are summarized. Section III gives a short introduction to the agile practices studied. Section IV specifies the research method and in section V the results of the analysis are presented. Conclusions and suggestions for further work can be found in section VI.

## II. Related work

A literature survey [5] on agile software development in regulated environment showed that most publications have concerned the avionics sector, in most cases regulated by DO-178B. Of the agile methodologies reported to be used, XP was the most frequent. In most studies agile was combined with traditional plan-driven methods.

Vouri [7] made a detailed analysis of general agile values, principles and practices against general principles of safety-critical software development. He concluded that many features of agile development can be beneficial for creating truly safe systems, but agile values, principles and practices must be adjusted to fit in a safety environment.

Detailed mappings between requirements and agile practices, similar to this study, have been reported for the avionics sector. Chisholm [9] mapped all the requirements in DO-178B with the agile methods XP, SCRUM and Crystal and found that agile methods can be adapted to satisfy the DO-178B objectives. He recommends more upfront planning and design than what is usual in existing established agile methods. Another study by VanderLeest and Buter [4] found that most agile practices are compatible or easily adapted to suit DO-178B development. Subcontractor relationship, lockstep gateways, large projects and legacy code were identified as challenges to introduce agile in a regulated context. The study by Wils and Van Baelen [6], focusing on DO-178B, found that most agile principles can be easily adopted in the early phases of the project. However, when it is time for certification some principles such as refactoring and welcome requirement change must be abandoned according to that analysis.

## III. Agile practices studied

This section gives a brief introduction to the agile practices included in this study, which are the originally ones presented for XP [1]. We acknowledge that there are additional practices that are considered agile, and can be of great benefit. However, the practices from XP were chosen as they represent concretely described software engineering practices, as well as widely used and well studied.

### *A. The planning game*

The Planning Game describes how project stakeholders (the Business) and the Team plan and steer the activities jointly. The Business specifies what to do in the format of "stories" and prioritize them in a backlog. The Team estimates the time to complete stories and their own capacity (velocity). The scope of the next release can then be agreed on. The time to release is typically divided into short time-boxed iterations in which the stories are broken down into tasks and completed.

### *B. Small releases*

XP stresses the importance of delivering the software incrementally, as small releases, each giving a true value for the customer. Even if the system cannot be placed into real production for each release, XP claims that there is a value to demonstrate and get user-feedback (validation) often. It also makes the organization prepared for the ultimate releases.

### *C. On-site customer*

To be able to make decisions efficiently and to keep the work aligned with the wishes, customer should be co-located with and/or highly available to the development team, according to XP.

### *D. Test driven development*

Test driven development (TDD) means that developer always should write tests, and make sure they fail, before writing any code. In that way the developer is enforced to think about the interface (input and output values) and the testability before the actual implementation. Applying this practice correctly yields high coverage through automated tests, which are typically run several times each day. XP teams use the automated tests as a means to ensure that changes, either triggered by new requirements or Refactoring, does not introduce new defects.

The principle that tests are written before development also applies to higher testing levels (integration and system level). This is often referred to as acceptance test driven development in the agile community. This means that tests are specified and normally automated before the implementation and the integration is performed.

### *E. Simple design*

In its essence, Simple design means that we should only design for what we need right now, i.e. the requirements to be implemented up to this iteration. This makes it possible to deliver value to the customer earlier and makes the design easier to understand and maintain, according to XP. When it is time to fulfill more requirements, the team must accept the fact that restructuring can be needed.

### *F. Refactoring*

Refactoring essentially means to improve the source code. After writing new code the developer should change the code to make it simpler, for instance remove

duplications. XP teams must also accept the fact that occasionally larger reorganization of the source code (even the architecture) is needed to accommodate new requirements.

### G. Pair programming

In Pair Programming, two developers sit together to solve a programming task. They discuss the design and then one of them writes the code while the other person reviews the code and thinks about the overall design and need for additional testing.

### H. Collective Ownership

Collective ownership means that anyone who that finds a way to add value to any part of the code is obliged to do so. No code is owned by an individual programmer. This makes changes efficient and reduces the risk that code could diverge [1].

### I. Continuous Integration

To find integration problems early it is important to synchronize with other team members often. In XP the code is integrated and built automatically several times a day. The automated test suites created as a part of test driven development are also run frequently to ensure that any integration problem is found and fixed as early as possible.

### J. Metaphor

XP stresses the importance of finding a metaphor for the system to make it easier to understand and explain the system. Beck [1] means that this to a high degree replaces what we ordinarily call architecture. In a safety-critical system there is normally a physical system that can be like a metaphor at the system level. These terms can be reused in the software model. For instance, to reason about and explain safety features in the software we can refer to well known physical safety arrangements like guards (to protect from invalid input) and firewalls, even though they sometimes are implemented as software elements.

### K. Coding standards

In conjunction with Pair Programming and Collective Ownership, it is important that the team agrees on common coding standards. Besides ordinary requirements on coding styles, XP emphasizes simplicity and ability to communicate as important parts of the coding standard.

### L. Sustainable pace

To be able to sustainably deliver high quality software, XP stresses the importance of not overworking the people involved. For pair programming it is also important to have common working times within a team.

## IV. Research method

This study was undertaken by asking the following questions for each agile practice as described in section III:

1. Are there any requirements in EN 50128 in conflict with the practice?
2. Are there any requirements in EN 50128 supported by the practice?
3. How can we adapt the XP practice for a team working under EN 50128 regulations?

The analysis was restricted to requirements in Clauses 4 to 7 of the standard, including references to other normative annexes in the standard. Clause 4 specifies the high level requirement on how to assign safety integrity levels (SIL) and how to apply the other requirements. Clause 5 defines the requirement of the management, organization and roles, while clause 6 is about software assurance. The requirements of the life-cycle, including documentation, for generic software development is in clause 7. Clause 8 about application-specific adaption and clause 9 about deployment and maintenance were excluded from the detailed analysis.

All requirements from Clause 4 to 7 were put in the rows of an Excel sheet with extra columns for each of the 12 agile practices. For each requirement and practice pair (cell) it was then noted whether the requirement was related with the practice in question or not. For each relation we then analyzed whether the practice supports and/or is in conflict with the objective of the requirement. Based on the results for each practice, possible ways to adapt the practice were suggested to fit in an EN 50128 regulated environment.

To validate the analysis, the results were reviewed by two independent researchers. When discrepancies were found, either between the original analysis and the reviewers, or between the reviewers, a discussion was held. The reasoning and results of the discussion was then added to the results of the analysis.

## V. Results and analysis

This section presents the results of this work, summarized in Table I.

### A. Test driven development (TDD)

The practice that developers write the tests themselves seems to be problematic in a regulated environment. According to EN 50128 it is the responsibility of the tester to specify the tests (req. 5.1.2.2) and the implementer and tester must be separate persons at all SIL levels (req. 5.1.2.10-12). On the other hand, test driven development supports the requirements that tests should be highly automated (req. 6.1.4.5) and that the source code shall be testable (req. 7.5.4.3).

Nothing in the standard prevents the implementer from writing own test scripts. However, to be accepted the tester must specify in more detail what to test at different levels. Applying structured test design techniques well as required by the standard is often beyond the knowledge of the implementer. An agile way of solving this is to let the tester specify the tests, but have the implementer to create the scripts for it, in a pairing session. The tester reviews the test scripts and writes the test specification mostly by referring to the scripts.

### B. Pair programming

Pair programming supports the requirement that the software source code shall be readable, understandable and testable

TABLE I: SUPPORTING AND PROBLEMATIC FEATURES OF AGILE PRACTICES IN AN EN 50128 REGULATED ENVIRONMENT

| Agile Practice | Supporting requirements | Problematic requirements |
|---|---|---|
| Test Driven Development | Testable code<br>Automated tests | Independence of tester<br>Tester specifies tests |
| Pair Programming | Source code readable and understandable | - |
| Planning Game | Taking iterations into account<br>Traceability | Details of requirements<br>Change management |
| On-site customer | Validation | - |
| Continuous Integration | Controlled test environment<br>Automated dynamic verification | - |
| Refactoring | Simpler, readable and maintainable source code | Risk to invalidate verification and validation |
| Small releases | Validation | High burden for formal certification each time |
| Coding standard | Coding standards required | - |
| Metaphor | Architecture and design simple and understandable | Not sufficient, too ambiguous |
| Simple design | Suitable design method<br>Balanced size and complexity of source code | - |
| Sustainable pace | - | - |

(req. 7.5.4.3). Code is reviewed continuously and the person that does not code can ensure that code is aligned with code conventions, requirements and tests. The other person can also help to answer questions that arise by looking at documentation, asking customer, tester and colleagues. The informal review performed in pair programming may not be accepted as a complete replacement for formal independent reviews. However, an additional review is probably needed and as pointed out in [4], "overhead of the extra review is likely to pay for itself by catching issues before they flow to later expensive stages". As mentioned above the tester can also occasionally pair with implementer to work out tests.

### C. *Planning game*

The EN 50128 standard does not preclude iterative approaches. In fact, the standard states that "the lifecycle model shall take into account the possibility of iterations in and between phases" (5.3.2.2). However, the XP practice to use simple paper cards to document requirements (stories) is unlikely to be accepted. Paper cards can be used as a visual tracking mechanism but requirements must still be placed in documents or in a tool as required by the standard. An agile way of doing this is to incrementally fill in the documents with detailed requirements, test cases and design relevant to the current increment. In an agile team the requirement manager, tester and developer work closely together and communicate frequently, which make coordinated and parallel updates in several artifacts efficient. Tasks to create and update documents and doing reviews are planned, estimated and accepted by team members with appropriate responsibilities, just like ordinary programming tasks.

Traceability between requirements, tests and source code is also an important aspect of the standard. By working with one or a few features at a time, working together and updating documents concurrently, traceability can actually be facilitated by working in an agile way. Since tests and source code are written only to fulfill the current requirements it should be straightforward to follow changes in the version control system by associating change sets with references to the requirements or tests in question. By following the history log it can then be determined which source code files (and which lines of source code) are associated to a specific requirement. By working with documents and other artifacts in a similar way and in the same version control system traceability can be enhanced.

XP stories usually capture only the functionality that too in vague terms. However, safety-critical systems have a lot of non-functional requirements as well (on performance and safety for instance). To be able to estimate a story more detailed requirements associated to the stories to be implemented are likely needed before an iteration starts. Well-structured requirement engineering is also demanded by the standard.

A big concern with all iterative methods is change management. When developing systems incrementally major changes are introduced in most iterations, which may invalidate previous work on assurance. According to the standard impact analysis must be performed and documented for each change, at least after deployment. Agile remedies for this are to include impact analysis workshops in each iteration and to automate the assurance activities as much as possible. It is also possible to postpone some formal assurance activities to near the end of a formal release period.

### D. *On-site customer*

No requirement in EN 50128 was found to prevent a customer or end-user (the Business role) to participate actively in the team work. By having a customer near the team validation is very much facilitated.

The person(s) playing the Business role varies depending on the size of the project. In small projects, such as development of a qualified tool, the end-user of the tool (e.g. test engineer) can play this role. In very small projects hardware designers are preferably included, or at least closely associated with the software team. In large projects the teams must necessarily be split into smaller ones regardless of agile or plan-driven. Then the Business role can be played by a member of a system or integration engineering team, which takes responsibility for the overall system design and integration.

### E. Continuous integration

No requirement in EN 50128 was found to be in conflict with the practice to integrate often. By running automated tests often the time to find problems is shortened. As part of periodic builds automated static checking and other verification techniques can be run as well. It is also very advantageous if newly built versions can be automatically deployed and tested on target hardware as part of periodic builds.

### F. Refactoring

On small scale and short-term (within an iteration and for new code) refactoring can be an effective mean to create simpler, more readable and more maintainable code. However, the cost of this will rise as soon as it requires changes, not only to the code, but changes and reverification and revalidation of associated artifacts. Therefore it is important to not create detailed design documentation too early (or generate it automatically). Some more up-front architecture than recommended by XP might be needed to avoid costly large-scale refactoring later. In addition, before changing the source code, a change request, followed by an impact analysis must be performed.

### G. Small releases

No requirement of EN 50128 was found to be in conflict with the practice to deliver and demonstrate functionality frequently. However, the extensive requirements on documentation, verification, validation and assessments make it costly to set systems in real production. On the other hand, by frequently integrating and demonstrating the system validation becomes stronger. This means that the risk to create solutions not suitable, even though they fulfill the requirements is reduced.

### H. Coding standard

Coding standard is a mandatory requirement in EN 50128 (7.3.4.25). Collective ownership and Pair programming enforce standards to be followed.

### I. Collective code ownership

Collective code ownership was not found to be in collision with any requirements of EN 50128 per se. However, all developers must be aware of that changing code might require costly reverification.

### J. Simple design

The practice to create as easy solutions as possible is in line with the requirements on suitable design methods (7.3.4.28) and balanced size and complexity of the developed source code (7.3.4.2). However, as discussed for Refactoring already, making too simple designs early can require a lot of rework later on to accommodate the requirements for the next iteration.

### K. System metaphor

The idea to use metaphors to describe the software design is not in conflict with any requirements. It supports the requirement that the design should be simple and understandable. However, it is not sufficient with general descriptions like that. Metaphors can mean different things for different persons. Moreover, the software architecture and design must still be documented in detail according to the standard. In XP it is generally recommended to wait with such documentation until the end of the development, but to enable verification and validation these documents must be created earlier and updated more often in a regulated environment.

### L. Sustainable pace

EN 50128 does not regulate the working time, but since stressed and overworked workers make more mistakes it is considered as positive for the general quality and safety of the final product.

## VI. Conclusions

This analysis shows that agile practices support some of the objectives and requirements of EN 50128. However, most practices must be tailored to fit in a regulated development environment. Nor do these practices offer a complete process, since many required documentation and assurance activities are not incorporated. These results are in line with the results from studies of the avionic domain regulated by DO-178B [4, 9].

In summary, the agile practices studied have a potential to make development more efficient by reducing the distance between customers, developers and testers. By using short cycles and frequent integration, tracking of progress and problems can be enhanced, but change impact analysis can be problematic. A big challenge is also to find an effective way of working with all the required documentation in an incremental way, creating and updating it only when it is needed, possibly using automated procedures. Obviously, the heavy requirements of documentation will make the work less agile.

Since the results in this analysis are not yet backed up by any empirical evidence, these results can only serve as an initial guidance for an organization considering agile practices in the development of railway software applications. Table I summarizes the strong and weak features of agile practices, which must be matched to the need of the particular organization. In addition, more general considerations on introducing agile practices in the organization must be made as described in [8]. It is also important to work closely with the quality assurance department and the assessor to ensure that the new practices are documented thoroughly. Otherwise, the project may fail the audit as described in [3]. For assessors, Table I and the discussion can give a hint on what to look for when facing an agile environment.

Further work includes proposing a more detailed agile process compatible with EN 50128 and assessing the process or some of the agile practices in a real pilot project developing software for the railway sector. In analogy with the recommendation for the avionic sector in [4] such agile pilot study can start with the development of a qualified tool and then further extended to a small-scale railway software development project.

## ACKNOWLEDGMENT

This work has been partially funded by the Swedish Knowledge Foundation (KK-stiftelsen) [10] and the EU-Artemis funded SafeCer project [14].

## REFERENCES

[1] K. Beck, “Extreme Programming Explained: Embrace Change” 1st ed., Addison-Wesley. 1999

[2] CELENEC, EN 50128 Railway applications. Communication, signalling and processing systems. Software for railway control and protection systems. 2nd ed. June 2011

[3] M. Poppendieck, “XP in a Safety-Critical Environment”, 2002. Cutter IT.

[4] S.H. VanderLeest and A. Buter, “Escape the waterfall: Agile for aerospace”, Digital Avionics Systems Conference, DASC '09. IEEE/AIAA 28th, Oct. 2009

[5] O. Cawley, X. Wang and I. Richardson, “Lean/agile software development methodologies in regulated Environments . State of the Art”, Proceedings of Lean Enterprise Software and Systems, Springer, 2010, p. 31-36

[6] A. Wils and S. van Baelen, “Towards an agile avionics process”, AGILE project report vol:D.2.12, ITEA-AGILE consortium. February 2007

[7] M. Vuori, “Agile development of safety-critical software”, Tampere University of Technology. Department of Software Systems. Report 14, 2011

[8] A. Sidky and J. Arthur, “Determining the applicability of agile practices to mission and life-critical systems”, Proceeding SEW '07 Proceedings of the 31st IEEE Software Engineering Workshop , 2007, p. 3-12

[9] R. A. Chisholm, “Agile software development methods and D0-178B certification”, Royal Military College of Canada, 2007

[10] KK-Stiftelsen: Swedish Knowledge Foundation, http://www.kks.se (Last Accessed: September 2012)

[11] IBM, Rational Unified Process, http://www-01.ibm.com/software/rational/ (Last Accessed: September 2012)

[12] Manifesto for Agile Software Development, http://agilemanifesto.org/ (Last Accessed: September 2012)

[13] S. Nerur, A. Cannon, B. VenuGopal and P. Bond, “Towards an understanding of the conceptual underpinnings of agile development methodologies”, Agile Software Development, Springer, 2010, p.15-29, doi: 10.1007/978-3-642-12575-1_2

[14] SafeCer project, www.safecer.eu (Last Accessed: October 2012)