Agile
Lean software development for avionic software

Emmanuel Chenu

Agile & Lean software development for avionic software

Emmanuel Chenu, Thales Avionics
emmanuel.chenu@fr.thalesgroup.com
Issue 1.1, 12/2011

Author

Emmanuel Chenu is an advocate and user of eXtreme-Programming, Scrum and Lean for the development of high-integrity embedded software. He is a software development coach at Thales Avionics in Valence. He is passionate about his field and likes to share his experience with others: he is a frequent speaker at Agile and Lean conferences, blogs and teaches pragmatic software development at ESISAR.

Abstract

We, Navigation unit of Thales Avionics, are struggling with two main problems when developing high-integrity avionic software. Firstly, the activities we perform to fulfill the mandatory safety objectives are expensive and time consuming. Secondly, our typical product integration is a "big-bang" integration. The second situation aggravates the first, and both situations are aggravated by the fact that our customers require that we develop fast. Fortunately, a common solution is emerging on several of our projects since 6 years: practices inspired from Agile software development and Lean. These two methods provide a efficient combination of rigor, speed of execution and continuous improvement.

Even though we face both problems simultaneously, they will be presented in sequence following a Plan-Do-Check-Act pattern. In sequence, each problem will be described with its impacts for the customer and the supplier. The root causes will be identified and the plan to solve the problem will be established. Then, the experiments performed according to this plan will be described. Lastly, the results of these experiments will be measured and the lessons learnt will be provided. This paper assumes that the reader is somewhat familiar with Agile software development and Lean. However, the minimal prerequisites are given hereafter before the Plan-Do-Check-Act case studies.

Agile software development and Lean in a nutshell

Agile software development is an umbrella term for software development methods that share the following values: individuals and interactions are valued more than processes and tools; working software is valued more than comprehensive documentation; customer collaboration is valued more than contract negotiation; responding to change is valued more than following a plan. At a first glance, Agile software development is far removed from how we have traditionally produced life-critical products. Our industry is more often characterized by detailed and heavily-tooled processes; by comprehensive and voluminous documentation; by contracted milestones, costs and scope; by planning and following plans.

The most famous of the Agile methods are eXtreme-Programming (XP) and Scrum. XP can be considered as a set of simple, yet interdependent organizational and software-engineering best practices. To reach maximal efficiency, they are combined and taken to extreme levels. These practices include frequent releases in short development cycles, business-value-oriented releases, systematic and automated testing, early and frequent automated integration, refactoring, working in pairs, simple design and coding standards.

Scrum is a lightweight pragmatic project management framework. A self-organizing team develops increments of shippable product in short iterations. The iterations implement the highest priority features first. The customer prioritizes the features by business value. Unlike XP, Scrum does not address engineering practices. Happily, Scrum fits in smoothly with the disciplined and rigorous engineering practices of XP.

Lean is a production system that considers the expenditure of resources for any goal other than the creation of value for the end customer to be wasteful, and thus a target for elimination. In a more basic term: "*more value with less work*". This is a variation on the theme of efficiency based on optimizing a continuous flow of value creation for the customer. The production flow is visualized in order to reveal problems. Problems are assessed

immediately to maintain the flow. Then, the problems are fully solved one by one in a methodic way in order to improve the companies standards. The practice of Lean in software development has revealed that Agile methods are an appropriate solution to deal with the problems of big-bang integration, unknown or unstable requirements, high bug-rates and challenging delivery milestones.

Costly safety-related activities are a problem

The DO178B is a document dealing with the safety of software used in airborne systems. Proof of airworthiness may be obtained after a series of audits performed by the regulation authorities: the SOI #1, #2, #3 and #4 milestones. To successfully pass these audits, the developer must prove that he has performed activities to fulfill the DO178B objectives. The number of objectives to satisfy depends on the level of criticality of the software. The level of criticality is given to an avionic software according to the impact of a failure on the flight's safety. The levels range from "*A. Catastrophic: may cause a crash*" to "*E. No effect*". As the level of criticality increases, so does the number of objectives to satisfy. They concern the software life-cycle processes.
Safety analysis is difficult to perform for software because it is not feasible to assess the number and the different kinds of software errors. An acceptable means for ensuring the safety of software is to impose rigor on the process used to build it. The requited level of airworthiness of a software program will be obtained by a given level of rigor in the development process, to prevent with enough confidence any remaining bug.

One of our main problems is that the activities we perform to fulfill the objectives of a DO178B software development are expensive and time consuming. We perform many of these activities late our projects and in big batches. The processing of big batches is laborious, does not reduce complexity and does not provide early feedback. This compromises the reliability and the efficiency of the activities we perform. This phenomenon gets worse for big batches that we process late in the project when we lack both time and money.
For the customer, the impact of this problem is a loss of confidence in the reliability of the safety-related activities. Also, the final delivery is over schedule as we are struggling with big batches of work and rework.
For the supplier, the impact of this problem is that big batches are not efficient. Also, part of the safety-related activities is performed late. Late feedback implies more rework. Little efficiency and late rework have a negative impact on costs and milestones.
There are several root causes to this problem. Firstly, the customer requires non-certified versions of the software (SW) early in the project. To deliver within the ambitious milestones, we set aside big batches of safety-related activities for the end of the development. Secondly, the verification activities (mainly tests and reviews) must be performed on the final version of the SW package. This also sets aside big batches of work for later. Thirdly, unlike the SOI #1 & #2 certification milestones, the SOI #3 milestone comes late in the project. This does not encourage to perform the related activities early. Lastly, we have been practicing the single-pass V life-cycle (also known as the waterfall model), where activities are performed in big batches. This process is often used in avionics because it maps closely to the SOI #1, #2 and #3 milestones.
Our plan to solve this problem has been inspired from Lean: it consists in reducing the cycle-time. This reduces the size of the batches of activities. This also brings us to perform activities as early and often as possible, providing much needed feedback.

According to this plan, we have changed the single-pass V life-cycle for incremental development in short monthly iterations. This cuts big batches into smaller one-month batches. Each one-month batch is processed iteratively. For each one-month batch (or iteration), the activities are performed at the most appropriate moment (just-in-time): requirements, design, code, tests, integration, verification, traceability and documentation. Many safety-related activities are performed within each iteration. Each iteration provides an increment of working product and an increment of proofs that safety-related activities were performed. Also, we have practiced eXtreme-Programming (XP) and have noticed it brings value for safety. Indeed, when practicing Test-Driven-Development (TDD) tests are systematic, self-checking and automated. Therefore, the test campaigns are complete, repeatable, objective, reliable and less expensive. There are two levels of such tests: "black-box" high-level tests (checking high-level requirements) and "white-box" low-level tests (checking low-level requirements). The Test-First practice of TDD enforces the independence of tests versus code, as the tests are specified before the code is written. Then, automated continuous integration ensures that the latest version of the SW is always fully, repeatedly, objectively and inexpensively verified by all these tests. Lastly, pair-programming installs systematic reviews of products (requirements, tests, code ...). Verification activities will again be performed on the final SW package. However, we expect this last big batch to be processed quickly as the great majority of problems has already been revealed and corrected by performing the safety-related activities within each iteration.

Such practices applied on a large scale DO178B level-A project have enabled us to measure encouraging results. The cycle-time has been reduced from one year to 20 days. Nine incremental versions have been delivered on

schedule to the customer in a 3-year time-frame. These early deliveries have not comprised the safety-related activities. Indeed, the SOI #1 and SOI #2 safety milestones have been successful on first pass. SOI #2 has even provided early feedback on SOI #3 related activities as the tests of the requirements were available for inspection. With 99.9% in-process structural coverage by tests, only 0.05 defects per thousand lines of code (Kloc) have been detected by the customer on the early versions. This is greater than a tenfold improvement. Therefore this experimentation has raised the level of quality. We'll be able to provide more conclusions concerning safety issues and costs next year.

Even though our problem is not yet solved, we have learned that continuous flow reduces batch-size and complexity. It provides early feedback and raises the level of quality. Safety-related activities are more reliable and the regulation authorities are receptive to this positive change. Furthermore, we believe a more continuous cooperation along the project with the regulation authorities would enhance this tendency. Lastly, we have learned that rigor and speed of execution are compatible and combine efficiently.

"Big-bang" integration is a problem

Our second main problem is that our typical product integration is a "big-bang" integration. Integration is critical as our projects comply to planned costs and milestones until integration starts. Then, they systematically get late and beyond expected costs.
For the customer, the first impact is late deliveries. Then, late customer change-requests may not be taken into account as the product is too unstable to accept change. Also, the customer looses confidence in the product as the bug-rate of non-final versions turns around 10 defects per Kloc. For the supplier, the impact is long and costly product integrations worth at least 30% of the development budget. We enter the "tunnel-effect" as we loose visibility on progress and costs because we do not manage to converge on a working and stable product. The second impact is that the following milestones must be postponed in terms of months, implying longer developments.
We have identified several root causes to this problem. Firstly, our cycle-time is too long: typically 1 to 2 years. Therefore, integration is a late big batch of work. Feedback on quality is available too late, implying massive rework. This is a known flaw of the single-pass V life-cycle. Secondly, the SW is too dependent on its working environment: the hardware (HW) and the real-time operating system (RTOS). When the HW and the RTOS are developed in parallel to the SW, the SW may not be run, tested and integrated until the working environment is available. Lastly, the black-box tests are difficult to automate because of the complexity of the working environment. Therefore, they are run too rarely and late in the project.
Our plan to solve this problem has been to reduce the cycle-time with short monthly iterations. Also, we have planned to decouple the SW from the HW and the RTOS. This enables to integrate early and often to converge on a continuously assembled and working product.

According to this plan, we have changed the single-pass V life-cycle for the practice of incremental development in monthly iterations. Each monthly increment of functionality is integrated into a working product. The phases of our previous waterfall model have become activities linked in a continuous flow of value creation. The flow starts and ends at the customer level. Requirements define the added value for the customer. Acceptance tests check that the expected value is indeed added into the product and that it is still working. Before, the activities were sequentially performed in big batches on the whole product scope. Now all the activities, including integration, are performed at each iteration on incremental fractions of the product scope. We have used Value-Stream-Mapping to identify the minimal required steps necessary to add value into the product. These steps enable to create an increment of product with proofs that safety-related activities were performed. These steps do include the integration of the increment of functionality into the product. This has standardized our current best way of adding value into the SW. Then, we have used Kanban-inspired task-boards to pull the development of the requirements by highest business value, to perform the activities (including integration) at the most appropriate moment (just-in-time), to level the amount of work-in-progress and to visualize problems in the flow.
We have practiced TDD. It consists in the following systematic steps: start by writing a test before writing (or changing) the code; run the test and see it fail; change the code until the test succeeds; improve the code using the tests as a non-regression tool. When practicing TDD, the tests are automated and self-checking. Therefore, we have been running the tests very often at no price. We have been using two levels of tests: customer acceptance "black-box" tests and developer "white-box" tests. In fact, the unit tests were the early users of the code. Therefore, the code was designed to be used and tested. This naturally decoupled the code. Indeed, systematic unit testing has led to an architecture where the core functionality was separated from the HW and the RTOS. The remaining necessary dependencies were isolated. Combined with an Object-Oriented programming language, TDD enabled to run the test suites on a development machine with mocks and stubs of the HW and the RTOS without changing the system under test thanks to the Dependency Inversion and

Dependency Injection patterns. Therefore, 99% of the code was tested and integrated well before the HW and RTOS were available. We experienced progress before hardware. The very same tests were then run on the target when it was finally available.
To prevent bugs and integration issues even further, we have been practicing Design-By-Contract (DbC) to "grow" foolproof code. Indeed, the code may not be used in any other way than required by its preconditions. The routine preconditions and postconditions are dynamically checked by assertions in code when it is run by the tests.
By practicing short-looped continuous integration, all these self-checking tests are automatically run when the code base is modified. Therefore, several times per day, the latest version of our SW is always fully, objectively, repeatably and inexpensively tested. This practice grows and maintains an assembled and working SW product.

Such practices have revealed encouraging results.The cycle-time has been reduced from one year to 20 days. Therefore, integration is no more a late big batch of work. It is now an activity performed early and very often within each iteration. In a 3-year time-frame, 9 versions of the product have been delivered on schedule to the customer. The SW has already successfully passed its first flight-tests. The number of defects detected by the customer and by our teams has been reduced more than tenfold to reach 0.1 problems per Kloc. The cost of product integration has been decreased from 30% to nearly 5% of the project budget.

Thanks to these experiments, we have learned that incremental development in short iterations with continuous integration considerably reduces the development cycle and greatly reduces the dependencies of the SW on its working environment (HW & RTOS). This enables to integrate early and very often. As a consequence, the product is always integrated and working. Furthermore, we have learned that the absence of HW and RTOS, once a problem, has become an asset. Indeed, this situation requires a design where problems such as core functionality on one hand and interfacing the HW and RTOS on the other hand are clearly and cleanly separated. The core functionality is fully tested on a development machine and the interfacing of the HW and RTOS is tested on the target. Thanks to the trust the developer has built into his fully tested code, he knows that the problems he encounters on the target now only concern issues of real-time, multi-threading, limited resources and improper interfaces with the HW or the RTOS. This clear separation between core functionality and interfacing the HW and RTOS now reveals opportunities for reuse and porting.

Conclusion

So, a shift of practices towards Agile SW development and Lean helps us to fulfill safety objectives with improved reliability, better efficiency and higher quality. Also, we have measured great improvements regarding product integration and testing. These improvements have enabled us to deliver within ambitious milestones. These large-scale experiments on live projects have convinced us that rigor and speed of execution are compatible and combine efficiently. In fact, our two initial problems may be considered as one same problem: processing large batches of activities. The common solution is to reduce batch-size and to perform activities early and often. Hopefully, the next steps of the certification of our SW will confirm these encouraging results.

Issues

- 0.0, 11/2011: Initial issue.
- 0.1, 11/2011: Issue taking into account remarks from Julien Duquenne and changes inspired from the design of the presentation slides.
- 0.2, 11/2011: Issue taking into account remarks from Isabelle Roth.
- 0.3, 11/2011: Issue taking into account remarks from Nicolas Blanpain and Pascal Fortin.
- 1.0, 12/2011: Issue following the 2011 Certification Together International Conference presentation the 1st of December at Toulouse. For the ERTS2 conference, the following topics are added: a summary of Agile and Lean, Design-By-Contract and the core-functionality vs HW and RTOS separation.
- 1.1, 12/2011: Issue taking into account remarks from pascal Fortin.