Agility and Lean for Avionics

First Issue, 01/04/2009

Emmanuel Chenu
emmanuel.chenu@gmail.com

ABSTRACT
"Would you trust your life to a flight software grown[1] by practitioners of Extreme-Programming?" Almost surprisingly, agile software development has successfully brought value to avionics. This article will introduce the difficulties encountered in this particular industry segment and how Extreme Programming and Scrum have greatly helped to deal with some of them. We will then consider how these agile practices contribute to the implementation of most of the principles of Lean in order to grow high-integrity products of value while reducing costs.
This paper has been written in preparation for a conference and roundtable organized by AdaCore[2] entitled "The Lean, Agile Approach To High Integrity Software"[3].
Pictures and references to companies have been removed from this issue for publication on the web.

ASSUMPTIONS

In writing this article, I assumed that the reader is somewhat familiar with Agile Software Development, Extreme Programming, Test Driven Development, Continuous Integration, Pair Programming, Design By Contract, Scrum and Lean. However, the minimal prerequisites are given hereafter or when needed.

ASSUMPTIONS / AGILE

Agile software development[4] 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 pretty far removed from how we have traditionally produced life-critical products. Our industry is more often characterized by processes, tools, documentation and following plans.

The most famous of the agile methods are Extreme Programming and Scrum.

ASSUMPTIONS / AGILE / EXTREME PROGRAMMING

Extreme Programming[5] can be considered as a set of simple, yet interdependent organizational and engineering best practices. To reach maximal efficiency, they are combined and taken to extreme levels.

1 I like the metaphor saying that software is grown and not built. The product grows as it is iteratively enriched with increments of functionality. Such an approach to software development requires the constant care of a gardener weeding and tending his garden. As the saying goes: *"software is soft and buildings aren't"*. For more on software development metaphors, read http://www.codinghorror.com/blog/archives/000987.html
2 http://www.adacore.com/
3 March 26, 2009 http://www.adacore.com/home/ada_answers/lectures/lean-event/
4 http://agilemanifesto.org/
5 http://www.extremeprogramming.org/

For example, code reviews are a best practice, so XP[6] recommends **Pair Programming**. All production code is written by pairs of programmers working together at the same workstation. One types in code while the other reviews each line of code as it's typed in. The two programmers switch roles frequently.

Tests are a best practice, so XP recommends **Test Driven Development**. It consists of short cycles based on the following steps: write failing automated self-checking tests that define desired changes or new functionality; write just enough code to make the failing tests succeed[7] and then refactor the code to improve the design. This technique is used at two levels: customer acceptance tests and developer unit tests. All tests are run automatically and repeatedly.

Design is important, so XP recommends **Refactoring**. Without changing the behavior of the system, the code is changed to improve the design. Improvements consist in removing duplicated code, making the code more explicit and simpler.

Simplicity avoids waste, so XP recommends **Simple Design**. The simplest working solution is implemented.

Integration is risky, so XP recommends **Continuous Integration**. A shippable program including the latest changes is always available. Programmers always work on the latest version of the software. Therefore they check in their code and integrate several times per day.

Customer feedback is important, so XP recommends **Small Releases**. Short iterations provide increments of shippable and usable products which allow customer feedback.

Customer collaboration and communication are important, so XP recommends **Whole Teams**. The team contains all the skills required to make the project succeed. The customer is part of the team. The team shares a common workspace.

XP is considered by some of its detractors as a framework for hacking and cowboy coding. On the contrary, we believe XP is a set of disciplined and rigorous practices.

ASSUMPTIONS / AGILE / SCRUM

Scrum[8] is a lightweight pragmatic project management framework. A self-organizing team grows increments of shippable product in monthly 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[9]. However, Scrum fits in smoothly with the disciplined and rigorous engineering practices of XP.

ASSUMPTIONS / LEAN

Lean is a production practice 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*".

---

[6] XP: Extreme Programming

[7] Please note that this practice, also named Test-First Programming, implies full code coverage by tests at all times.

[8] http://www.controlchaos.com/

[9] Some projects adopting Scrum fail because they forget to combine Scrum with serious engineering practices such as those of XP.

AVIONICS

In the avionics business, we develop airborne flight equipment. A failure in such a product may impact the safety of flight, making it a life-critical product. A great deal of the functionalities are performed by real-time, embedded software. They must be certified by a specialized organization to ensure the safety of flight. Apart from some difficulties shared with the larger, more general software development industry, this segment is confronted by the issues of **real-time embedded technology** and **safety-criticality**. Lets now consider each of these difficulties and how XP helps to deal with them.

REAL-TIME EMBEDDED TECHNOLOGY & AGILE

REAL-TIME EMBEDDED / SPECIFIC HW AND RTOS

The software runs on a specific hardware with a specific real-time operating system. The developers must deal with real-time requirements, multi-threaded computations and cope with limited processing and memory resources.

REAL-TIME EMBEDDED / TEST AND INTEGRATION / ISSUES

Often, the HW[10] and the RTOS[11] are developed in parallel to the software. In such a case, both are available late in the project and in limited quantities. Therefore our typical SW[12]/SW and SW/HW integrations were late and Big Bang-ish.
Testing at this stage was not efficient. It essentially consisted in debug sessions on the target. Moreover, it was a pity to check complex business algorithms on the non-ergonomic development environment of the HW target.

REAL-TIME EMBEDDED / TEST AND INTEGRATION / AGILE SOLUTIONS

The practice of XP's TDD[13] naturally decouples the code. Systematic unit testing leads to an architecture where the core functionality is separated from the HW and the RTOS. The remaining necessary dependencies are isolated. When combined with an Object-Oriented programming language, TDD enables to run test suites on a development machine with mocks and stubs[14] of the HW and the RTOS without changing the system under test thanks to dependency injection[15]. Therefore, the code is tested and mostly integrated well before the HW and RTOS are available[16]. The very same tests are then run on the target when it is finally available[17].

In fact, not having the HW and the RTOS, once a problem now becomes an asset. It requires to design an architecture where problems such as core functionality on one hand and the HW and RTOS on the other hand are clearly and cleanly separated. The core functionality (with the reusable business code) is fully tested on a development machine and the interfacing with the HW and the RTOS is tested on the target.

Thanks to the trust the developer has built into his fully tested code, he knows that the problems

[10] HW: hardware
[11] RTOS: real-time operating system
[12] SW: software
[13] TDD: test driven development
[14] Mocks aren't stubs: http://martinfowler.com/articles/mocksArentStubs.html
[15] Refer to http://en.wikipedia.org/wiki/Dependency_injection or http://martinfowler.com/articles/injection.html
[16] Read Progess Before Hardware: http://masters.donntu.edu.ua/2005/fvti/semisalova/library/progressbeforehardware.pdf
[17] When coding in Ada, building test suites using a native version of the embedded Ada runtime

he will encounter on the target will now only concern issues of real-time, multi-threading, limited resources and improper interfaces with the HW or the RTOS.

PROGRESS BEFORE HARDWARE[18]
Recently, we have developed a navigation software program for a customer who had no spare HW target to lend us during our development. Moreover, the customer did not want us to integrate our code onto his few hardware prototypes because his teams would have to code with one less board available. So, for 6 months we grew our code using TDD with stubs and mocks of the HW and the RTOS. Finally we came to integrate at the customer site. The SW/HW integration took 4 days and we discovered 6 defects. All concerned SW/HW interfaces and none concerned the code's functionality. The bug rate was 0.3 bugs per 1 KLOC. The customer was happy because he had one HW prototype unavailable for his teams for just 4 days.

REAL-TIME EMBEDDED / REUSE AND PORTING / ISSUES
We have never managed to reuse or efficiently port code that is heavily coupled to the HW or the RTOS. Porting an application from one target to another meant great costs.

REAL-TIME EMBEDDED / REUSE AND PORTING / AGILE SOLUTIONS
As TDD separates the core functionality from the HW and the RTOS, reuse and porting have now become common practice. With multi-project version control and Continuous Integration, we now grow products where 30% of the code is reused among products of different product lines and 60% of the code is reused among products of the same family.

library is a great way to ensure the tests may be run on the target without adaptation.

[18] « Progress Before Hardware » come from James Grenning. Read his paper: http://masters.donntu.edu.ua/2005/fvti/semisalova/library/progressbeforehardware.pdf

SAFETY & AGILE

The main concern in avionics remains safety.

SAFETY / LEVEL OF CRITICALITY

A level of criticality is given to avionics software according to the impact of a failure of that software on the flight's safety. The levels range from *"A. Catastrophic: may cause a crash"* to *"E. No effect"*.

SAFETY / GUIDELINES

Document DO178B, published in 1992, specifies the objectives to achieve during development in order to certify software for flight. Proof of airworthiness is obtained after a series of audits. They require the inspection of proofs of the activities carried out to support these objectives.

SAFETY / OBJECTIVES

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 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.

THE CLEAN WATER PIPE METAPHOR
If we use a metaphor where the pipe is the process and the water is the product, then the idea behind the guidelines is that a dirty pipe cannot deliver clean water. However, a clean pipe helps to deliver clean water

SAFETY / TRADITIONAL PROCESS

The DO178B defines objectives that must be satisfied but not the process to achieve these objectives. The objective-based nature of DO178B allows a great deal of flexibility in regard to following different styles of software life cycles. For example, the guidelines explicitly mention iterative and incremental[19] development. However, as the objectives concern the software life cycle, its activities, the activity entry and exit criteria and the proofs expected, the V model has historically been the common way to drive DO178B compliant developments. Also, software has been certified using this method so it is still used for this purpose.

SAFETY / ISSUES

A certification audit will check that the product exclusively contains code satisfying operational requirements. The implementation of all the operational requirements and all the code will have to be verified by tests. All the tests will be performed on the final airborne version of the program. A minor change in the code of the final version implies that all the tests be run again[20]. Also, all the products (requirements, architecture, code, tests, traceability) will have to be validated in a review process. These activities may imply a considerable amount of work.

SAFETY / AGILE SOLUTIONS

Fortunately XP's practices contribute to achieve safety objectives.
XP's incremental construction driven by operational scenarios enforces that the product will exclusively contain code implementing operational requirements.

[19] The process is **iterative and incremental** when activities are repeated to add usable functionality to a shippable product.
[20] This can imply a considerable cost if testing is performed manually by an operator.

XP's systematic acceptance testing will guarantee that the implementation of all operational requirements is successful.
XP's systematic developer testing will ensure that all the code is checked by tests[21].
XP's Test First Programming is a great way to enforce independence of tests versus code.
The full acceptance and developer self-checking tests are run automatically by the Continuous Integration tool when code is changed. This ensures at a minimum cost that the latest version of the program is always fully and repeatably tested.
Finally, XP's Pair Programming practice will ensure that all products are reviewed.

Therefore, the practice of XP helps to develop efficiently and brings value for certification.

AUTOMATED SELF-CHECKING TESTS ARE SAFER (AND CHEAPER)
Recently, a friend and I quickly walked through an open space where developers where running tests on a high-integrity software. We both noticed that all the tests procedures where run manually. We also noticed that an operator was distractedly typing a SMS message on his cell phone while running a manual test. I am sure that the manual test procedure was boring. I am sure that the operator already ran this test several times. I am sure that this test run was not used for certification purposes. However, my friend and I felt we would be more comfortable if the tests where automatically and repeatably run instead of manually performed by a distracted and tired operator.

SAFETY / BEWARE OF THE LEVEL OF FORMALISM REQUIRED
However, without any adaptation, XP’s set of by-the-book practices does not provide the formalism and the proofs required for certification inspections.

SAFETY / ADAPT XP
This is why we have a practice of XP tailored for our industry. The values and principles of XP remain perfectly compatible with safety issues. However, some practices need to be adapted. For example, the documentation must be considered as part of the product. Therefore, the documents must be incrementally written to be potentially shippable after each iteration. Here's an example of a practice we added.

SAFETY / REQUIREMENTS AND TRACEABILITY / ISSUES
We have noticed that when we have trouble certifying a software program, the issues mainly concern requirements and traceability.

SAFETY / REQUIREMENTS AND TRACEABILITY / AGILE SOLUTIONS
In XP, all the difficult activities are performed early and often. Reviews are performed continuously in pairs. Testing is continuously led by Test First Programming. Integration is performed often thanks to Continuous Integration. Architecture is done early and often through iterative and incremental development with continuous refactoring.
Requirements and traceability are an issue, so we've added the continuous traceability practice. As we work, we record the traceability links. Full traceability is continuously and automatically consolidated, quantified, checked and finally published. Therefore, full traceability is always available and issues are detected early and corrected early.

[21] I insist: TDD implies full code coverage by tests at all times.

TEAMS & AGILE

In avionics, we also face human resource issues.

TEAMS / ISSUES
We have problems finding developers experienced in software engineering, embedded software and avionics. Also, team members seem to lose some of their initial motivation on large, long-lasting projects that are punctuated by quite a lot of documentation and traceability. This feeling is accentuated by the historical mass-production culture and the CMMi which both tend to have interchangeable workers through the separation of thinking from doing.

TEAMS / AGILE SOLUTIONS
When you don't have all the experienced developers you wish you had, XP helps because its practices enforce training on the job. Multidisciplinary teams, working in pairs in a common workspace enable to speed-up training. Also, XP enforces personal motivation by the special importance granted to the individual, the human touch in programming, teamwork, team spirit and self-organizing teams.

CONCLUSION AND TRANSITION TO LEAN

Practice has proven us that agile development and XP in particular help to grow real-time embedded and life-critical software. However, these approaches to software development do not provide the level of formalism with the proofs required for certification purposes. Tailoring is required.
We have also experienced that the bottom-up agile methods match nicely with top-down Lean approach to product development.

AGILE & LEAN

Toyota develops and builds life-critical products. When you are driving your car at 130 kilometers per hour on the highway with the cruise control system on, you are trusting your life to a high integrity product. A failure in the system may endanger the safety of the ride.
Toyota has designed a philosophy for developing and manufacturing products. Westerners have named it Lean.
To develop products of value while reducing costs, we are trying to shift our industrial paradigm from mass-production to Lean. Agile software development, with XP and Scrum, enable us to implement most of the principles of Lean for our development projects.

LEAN / THE 5 PILLARS

The 5 pillars of Lean are Value, Value Stream, Flow, Pull and Perfection. Let's now consider how our agile practices help to implement most of these principles.

LEAN / VALUE

"Specify value from the standpoint of the end customer."
XP recommends having an on-site customer. Unfortunately, we cannot have a certifier, an aircraft manufacturer and a pilot in each of our project teams. So, as recommended in Scrum, our project teams have a Product Owner[22] who represents the customer. In our case an avionics expert represents the aircraft manufacturer and the pilot. Also, a dedicated quality assurance engineer represents the certifier.
The Product Owner expresses product value using requirements. He expresses his satisfaction criteria by providing acceptance tests.
The Product Owner prioritizes the requirements by business value. This list of prioritized requirements is Scrum's Product Backlog[23]. The priorities define the succession of development iterations that incrementally grow the product. Therefore the iteration plan maximizes the customer's return on investment.

LEAN / VALUE STREAM

"Identify all the steps in the value stream for each product family, eliminating every step that does not create value."
There is no agile practice that really helps here, though all XP practices implement the Simplicity value, which consists in maximizing the amount of things to not do. Therefore, we practice classical Lean Value-Stream-Mapping.

LEAN / VALUE STREAM / DO NOT SPUR A WILLING HORSE

In Value-Stream-Mapping, you have to learn to see waste. Waste is work that creates no value. For example, in our industry, we are terrorized by the fact that our program will not fit on the target due to excessive CPU load or excessive memory occupation. Therefore, many in our industry optimize prematurely their code. They optimize on fear and not on facts. This can be considerable waste, as the code may be good enough! The effort of optimizing is then completely lost. Even worse, the optimized code is more complex for no reason. Therefore, our practice is to not optimize prematurely.
However, we monitor performance early and often. Code will then be optimized on hard facts. We believe that it is far easier to make a correct program fast than it is to make a fast program

[22] The **Product Owner** is the person who is responsible for what the team builds and for optimizing the value of it. The Product Owner is responsible for maximizing the value of the product being developed while minimizing the risk. The Product Owner represents the stakeholders in the project.

[23] The **Product Backlog** is a prioritized list of functional and nonfunctional requirements and features to be developed for a new product or to be added to an existing product.

correct. To get hard facts, we automate CPU load measurement and code profiling in our development cycle and start these activities early.

How do we prevent premature optimization from creeping into the code? XP teaches us to practice Pair Programming. These continuous code reviews among pairs enable programmers to detect and discourage premature optimization.

DO NOT OPTIMIZE PREMATURELY
Navigator: *"We need to store those 30 booleans."*
Driver: *"Here's an integer."*
Navigator: *"Do you mean you want to store those 30 booleans on the 30 first digits of that integer?"*
Driver: *"Hey, isn't that clever and efficient?"*
Navigator: *"And you want to code operations to encode and decode booleans on the digits on an integer?"*
Driver: *"Just common stuff."*
Navigator: *"Tell you what, let's simply use an array of 30 booleans, run the tests and check-in the code. Tomorrow, well get the results of the nightly automated code profiling and this Friday we'll measure the CPU load and memory occupation on the target."*
Driver: *"What if we do not satisfy the resource requirement because of our code?*
Navigator: *"What if we stop programming because the financial crisis could prematurely end this project?"*
Driver: *"OK, I get your point."*

LEAN / FLOW
"Make the remaining value-creating steps occur in a tight and integrated sequence so the product will flow smoothly towards the customer."
To ensure a steady flow, we reduce the size of the batches of work. We settled for Scrum's regular 4-week time box, called a Sprint[24]. A Sprint is an iteration providing a potentially shippable increment of product. Iterative and incremental development in short regular cycles ensures continuous flow of value. To maintain flow, we try to fix the problems that slow it down.

Which issues in software development slow down the flow of value?

LEAN / FLOW / DESYNCHRONIZATIONS
For example, desynchronization works against flow.
Team members take a clean version of the software into their own private workspace to add functionality to it. They then do not disturb other team members with implemented changes. As soon as a developer changes something on the software in his private workspace, he is desynchronized from the mainstream software. There will be some amount of effort to push his changes back into the flow of the mainstream software. This effort is commonly called integration.

We minimize these desynchronizations, and therefore the integration effort thanks to 2 agile practices. Scrums' short daily stand-up meetings[25] enforce a steady flow by synchronizing the team members and their activities. Also, XP's Continuous Integration ensures a steady flow of tested, integrated and shippable code. Thanks to these two practices, teamwork and the product remain desynchronized as short as possible.

[24] A **Sprint** is an iteration in Scrum, normally of a one-month duration. A Sprint delivers a shippable increment of valuable product.
[25] A **daily stand-up meeting** is a short focused team meeting. Each team member answers to 3 questions: What did he do yesterday? What does he plan to do today? Is any impediment slowing down his work?

LEAN / FLOW / DEFECTS
Bugs slow down the flow as time is spent on debugging.

LEAN / FLOW / DEFECTS / PREVENTIVE ATTITUDE
A preventive attitude toward defects makes sure bugs least affect the flow. To prevent bugs from creeping into the software we use several agile practices such as XP's Test First Programming[26] and Pair Programming. Also, Design By Contract[27] helps us to grow foolproof code, as you can't use the code in any other way than required by its preconditions[28]. Finally, changes cannot be delivered into the project repository if the tests fail.

LEAN / FLOW / DEFECTS / PROGRAMMING LANGUAGE
An object oriented programming language with an xunit framework is a great asset for practicing TDD. The object-oriented features enhance testability as they enable easy mocking and stubbing. A programming language with embedded Design By Contract features such as preconditions and postconditions[29] checked at runtime is a great asset for growing fool proofed software.
Therefore, your programming language may help to have a preventive attitude towards defects[30].

LEAN / FLOW / DEFECTS / STOP THE LINE
Unfortunately bugs still manage to creep into the code. In order to maintain a steady flow, we practice Lean's “Stop The Line”[31].
Our Continuous Integration tool compiles the code and runs all the tests as soon as it detects a change in the mainstream code. If the build fails for any reason, such as a failed test, then an email is sent to every team member. Then, our main priority is to stop and fix the bug. Design By Contract also enforces «Stop the line», as the precondition and postcondition assertions[32] abort the software as soon as an assertion fails. To run the code, there is no other available solution than to fix the code.

LEAN / FLOW / DEFECTS / FIXING
A Continuous Integration system that continuously builds and tests the system and yells when it detects a failure is a practical way of stopping to fix problems. The automated build, the automated tests and the assertions in the code exercised by the tests are the failure detectors. This practice builds a culture of stopping to fix problems, to get quality right the first time.

---

[26] In TDD, tests are meant to prevent bugs and not to detect bugs.
[27] **Design By Contract** is a technique that focuses on documenting, checking, and agreeing to the rights (preconditions) and responsibilities (postconditions) of software modules to ensure program correctness.
[28] The **precondition** of a routine is what must be true in order for the routine to be called.
[29] The **postcondition** of a routine is what the routine guarantees as long as its precondition is true.
[30] Several iterations of practice have convinced us that AdaCore's Ada2005 package containing full object-oriented features, built-in Design By Contract and the Aunit test framework is a great asset for growing high-integrity software.
[31] Production stops if an abnormal situation arises. This prevents the production of defective products, eliminates overproduction and focuses attention on understanding the problem and ensuring that it never recurs. It is a quality control process that applies the following four principles: Detect the abnormality; Stop; Fix or correct the immediate condition; Investigate the root cause and install a countermeasure.
[32] An **assertion** is a predicate (i.e., a true–false statement) placed in a program to indicate that the developer *thinks* that the predicate is always true at that place. Several modern programming languages include checked assertions that are checked at runtime. If an assertion evaluates to false at run-time, an "assertion failure" results, which typically causes execution to abort. This draws attention to the location at which the logical inconsistency is detected. Assertions may be deactivated at compilation.

LEAN / FLOW / INFORMATION
Finally, we ensure a smooth flow of information inside the team by collocating multidisciplinary teams in a common workspace. The team uses visual control so no problems are hidden. Indeed, the common workspace is loaded with Kanban[33] charts identifying bottlenecks, Burndown Charts measuring progress and Blocking Boards revealing impediments.

LEAN / FLOW / CONCLUSION
Iterative and incremental development in short cycles and Continuous Integration are short cycled processes that ensure continuous flow to bring problems to the surface. Flow is maintained by stopping to fix these problems.

LEAN / PULL
"As flow is introduced, let customers pull value from the next upstream activity."
The features to implement are prioritized by the Product Owner in the Product Backlog. The iterations successively pull the highest priority features out of the Product Backlog to transform them into valuable software. Therefore, by prioritizing the features in the Product Backlog, the customer is pulling the development of his product. In fact, the incremental and iterative development of prioritized features is a pull system for product development.

When a feature is pulled out of the Product Backlog, all the activities carried out to transform it into valuable software are identified in the Value Stream Map and pulled by a Kanban system.

In TDD, failing acceptance and developer tests are a pull system for coding. Indeed, the failing tests are written first and then just enough coding is pulled to make the failing tests succeed. TDD can therefore be considered as a pull system for coding!

LEAN / PULL / CONCLUSION
Pull is implemented at several levels. Incrementally growing a product by developing the customer's highest priority business value first is a pull system for product development. Failing acceptance tests and unit tests pull coding[34]. Daily tasks are pulled by the Kanban system. These pull systems avoid overproduction.

LEAN / PERFECTION
"As these steps lead to greater transparency, enabling managers and teams to eliminate further waste, pursue perfection through continuous improvement."
At the end of each monthly iteration, we run a Scrum retrospective[35] meeting. During this meeting, the team and the Product Owner analyze the iteration and try to improve the development practices.

LEAN / LEAN HELPS
Lean has really helps us in two ways. Firstly, Lean contributes practices not available in agile software development such as value stream mapping, Kanban pull systems and stopping to fix problems. Lean also helps to communicate efficiently on our practices and their motivations. Management and customers are not necessarily receptive to vocabulary such as Agile, Extreme Programming, Scrum, ScrumMaster, Pair Programming, Stand Ups and retrospectives. The vocabulary seems awkward and intuitively and erroneously more fitted for hacking and cowboy

---

[33] A **Kanban** is a signaling system to trigger action.
[34] May I insist one last time? Test First Programming ensures full code coverage by tests.
[35] A **retrospective** is a time-boxed meeting after the Sprint Review when the Scrum Team reviews the just-finished Sprint. After reviewing everything that worked well and things that could be improved, the team defines several changes to how it will work together for the next Sprint.

coding. On the other hand, Lean has a lot of credit in the industry as it successfully grew from the industry.

CONCLUSION
We are not doomed to use predictive development processes to build critical or embedded software. Hard facts show that Agile Software Development helps in our field. Its practices implement most of the pillars of Lean to grow high-integrity products of value while reducing costs.
These best practices may be sorted in two main categories: organizational and engineering practices. They are interdependent and must be combined to be effective. Iterative and incremental development in short cycles will not succeed if you have not settled rigorous and disciplined engineering practices. Successfully adopting an Agile and Lean approach to growing software requires serious technical practices such as automation, TDD, Continuous Integration, Design By Contract and object-oriented programming. In Agile Software Development, in Lean and in high-integrity product development, technical excellence remains an absolute requirement.

ACKNOWLEDGMENTS
I would like to thank Jamie Ayre and Kathy Fairlamb from AdaCore, Galia Ladiray-Wiess from AKKA Technologies and Jean-Pierre Decoux from Orange Business Services who provided much needed feedback and corrections to this paper.

FURTHER READING

On Extreme Programming:

- *Extreme Programming Explained: Embrace Change, Kent Beck, ISBN-10: 0321278658*

On Scrum:

- *Agile Software Development With Scrum, Ken Schwaber and Mike Beedle, ISBN-10: 0130676349*
- *Agile Project Management With Scrum, Ken Schwaber, ISBN-10: 073561993X*

On Lean Software Development:

- *Lean Software Development: An Agile Toolkit, Mary and Tom Poppendieck, ISBN-10: 0321150783*
- *Implementing Lean Software Development: From Concept to Cash, Mary and Tom Poppendieck, ISBN-10: 0321437381*
- *Lean Software Strategies: Proven Techniques For Managers And Developers, Peter Middleton and James Sutton, ISBN-10: 1563273055*

On object-oriented programming and Design By Contract:

- *Object-Oriented Software Construction, Bertrand Meyer, ISBN-10: 0136291554*

On agile development and embedded software:

- *Embedded Extreme Programming: An Experience Report*[36] *- Nancy Van Shooenderwoert - 2004*
- *Effective Test Driven Development for Embedded Software*[37] *- Michael J. Karlesky, William I. Bereza, Carl B. Erickson - 2006*

[36] http://www.agilerules.com/articles/Embedded_Extreme_Programming_Experience_Report.pdf
[37] http://atomicobject.com/files/EIT2006EmbeddedTDD.pdf

- *Extreme Programming And Embedded Software Development*[38] *- James Grenning – 2002*
- *Progress Before Hardware*[39] *- James Grenning - 2004*
- *Embedded Test Driven Development Cycle*[40] *- James Grenning - 2004*

On agile development and critical software:

- *Towards An Agile Avionics Process*[41] *- Andrew Wils, Stefan Van Baelen – 2007*
- *XP in a Safety-Critical Environment*[42] *- Mary Poppendieck – 2007*
- *Launching XP Extreme Programming at a Process-Intensive Company*[43] *- James Grenning - 2001*

38 http://www.objectmentor.com/resources/articles/EmbeddedXp.pdf
39 http://masters.donntu.edu.ua/2005/fvti/semisalova/library/progressbeforehardware.pdf
40 http://www.objectmentor.com/resources/articles/EmbeddedTddCycle-v1.0.pdf
41 http://www.agile-itea.org/public/papers/agileavionics.pdf
42 http://www.poppendieck.com/safety.htm
43 http://www.objectmentor.com/resources/articles/XP-In-Process-Intensive-Company-IEEE.pdf