
Article:
Chapman, Roderick, White, Neil and Woodcock, Jim orcid.org/0000-0001-7955-2702 (2017) Viewpoints:What can agile methods bring to high-integrity software development? Communications of the ACM. pp. 38-41. ISSN 0001-0782

https://doi.org/10.1145/3133233

What can Agile methods bring to high-integrity software development?

Roderick Chapman,[1] Protean Code Limited and Altran UK, UK

Neil White,[2] Altran UK, UK

Jim Woodcock,[3] University of York, UK

Abstract *We discuss the issues and opportunities raised by Agile practices in the development of high-integrity software, based on the scientific literature, projects, and our own understanding of relevant regulatory regimes, standards, and markets. We consider Agile assumptions and where these conflict with high-integrity development. Conversely, we also consider opportunities where an Agile approach could be improved by the adoption of high-integrity practices.*

1 Introduction

There is much interest in Agile engineering, especially for software development. Agile's proponents promote its flexibility, lean-ness, and ability to manage changing requirements, and deride the plan-driven or waterfall approach. Detractors criticize Agile's free-for-all.

At Altran UK, we use disciplined and planned engineering, particularly when it comes to high-integrity systems that involve safety, security, or other critical properties. A shallow analysis is that Agile is anathema to high-integrity systems development, but this is a naïve reaction. Pertinent questions include:

- Is Agile compatible with high-integrity?
- Where is Agile inappropriate?

[1] rod@proteancode.com

[2] neil.white@altran.com

[3] jim.woodcock@york.ac.uk

- Do Agile's assumptions hold for high-integrity or embedded systems?
- Could high-integrity best-practice improve Agile?

We don't have all the answers, but we hope this article continues to provoke debate on this important topic.

1.1 Why bother with Agile at all?

We often encounter two myths regarding the "traditional" approach to high-integrity software development: that we somehow manage to perform a single-iteration waterfall style process, and that "formal" notations are not amenable to change. Neither ring true with our experience. As our projects develop, they have to absorb change and respond to defects just like any other. This led to an observation: your project is going to become iterative whatever you do, so you might as well plan it that way from the beginning. This lesson was put to good effect in the MULTOS CA project (Hall 2002), which initially planned for seven iterations, but delivered after thirteen, owing to a barrage of change requests. Nevertheless, it worked, *and* we were able to keep a substantial formal specification and all the other design documentation up to date as the project evolved. Knowing that "change" and "iteration" were at the heart of the Agile manifesto, we decided to see what we could learn and bring to future projects.

2 Background and Sources

Many consider Agile started with XP (Beck 1999), but its roots are much older. Many of XP's core practices were well-established long ago—their combination and rigorous practice was novel. A survey (Larman 2003) notes that both incremental and iterative styles of engineering were used in the 1950s. Redmill's work on evolutionary delivery (Redmill 1997) predicted many of the problems faced by Agile projects. (Boehm 2003) provides some useful insight, while the development of MULTOS CA (Hall 2002) compared Correctness-by-Construction with XP (Chapman 2003), showing that the two weren't such strange bed-fellows after all.

Lockheed Martin developed the Mission Computers for the C130J by combining semi-formal specification, strong static verification, iterative development, and a strongly Lean mind-set (Middleton 2005). Use of Agile has been reported by Thales Avionics (Chenu 2009), while SINTEF have reported success with SafeScrum (SINTEF 2015). A recent and plain-speaking evaluation of Agile

comes from (Meyer 2014), although he doesn't specifically deal with high-integrity issues.

3 Agile Assumptions and Issues

How do Agile's practices and assumptions match real high-integrity projects? Here are some of the most obvious clashes. For each issue, we start with a brief recap of the practice in question, then go on to describe the issue or perceived clash, followed by our ideas and experiences in overcoming it. Where possible, we close each section with an example of our experience from the C130J, MULTOS, or iFACTS projects.

3.1 Dependence on "Test"

Agile calls for continuous integration, with a regression test suit, and a test-first development style, with each function associated with specific tests. Meyer calls these practices "brilliant" in his summary analysis (Meyer 2014), but Agile assumes that dynamic test is the principal (possibly only) verification activity, saying when refactoring is complete, or when the product is good enough to ship.

The safety-critical community hit the limits of testing long ago. Ultra-reliability can't be claimed from "lots of testing." Security is even harder—corner-case vulnerabilities, such as HeartBleed, defy an arbitrarily large amount of testing and use. In high-integrity development, we use diverse forms of verification, including checklist-driven reviews, automated static verification, traceability analysis, and structural coverage analysis.

There is no barrier between these verification techniques and Agile, especially with an automated integration pipeline. We try to use verification techniques that complement, not repeat each other. If possible, we advocate for *sound* static analyses (tools that find *all* the bugs, not just some of them), since this gives greater assurance and reduces pre-test defect density. With careful consideration of the assumptions that underpin the static analyses (Kanig 2014), we can reduce or entirely remove later testing activities.

The NATS iFACTS system (Chapman 2014) augments the software tools available to air-traffic controllers in the UK. It supplies electronic flight-strip management, trajectory prediction, and medium-term conflict detection for the UK's en-route airspace, giving controllers substantially improved ability to plan ahead and predict potential loss-of-separation in a sector. The developers precede commit, build, and testing activities with static analysis using the SPARK toolset. Overnight, the integration server re-builds an entire proof of the software, populating a persistent cache, accessible to all developers next morning. Working on an isolated change, the developers can reproduce the proof of the entire system in

about 15 minutes on their desktop machines, or in a matter of seconds for a change to a single module. While Agile projects might have a "don't break the tests" mantra, on iFACTS it's "don't break the proof (or the tests)…"

3.2 Upfront Activities and Architecture

Agile advocates building what's needed now, using re-factoring to defer decisions. Refactoring must be cheap, fast, and limit rework to source code.

Our principal weapon in meeting non-functional requirements is *system* (not just software) architecture, including redundancy and separation of critical from non-critical. Such things can be prohibitively expensive to refactor late in the day. We need just enough upfront architecture work to argue satisfaction of key properties. We also do a "What If?" exercise to make sure that the proposed architecture can accommodate foreseeable changes.

The MULTOS CA project had some extraordinary security requirements, which were met by a carefully considered combination of physical, operational, and computer-based mechanisms. The software design was much simplified as a result of this whole system view. The physical measures included the provision of a bank vault and enclosing Faraday cage—hardly items that we could have ignored and then "refactored in" later.

3.3 User Stories and Non-Functional Requirements

For security and safety, we must ensure our specification covers all possible inputs and states. Agile uses stories to document requirements, but these sample behavior, with no completeness guarantee. The gaps between stories may contain vulnerabilities, bugs, unexpected termination, and undefined behavior. Meyer files user stories under "Bad and Ugly", and we agree.

For critical systems, we prefer a (semi-)formal specification that offers some hope of completeness. The C130J used Parnas Tables to specify critical functions. They seemed to work well—they were simple enough to be understood by system engineers, yet sufficiently formal to be implemented and analyzed for correctness.

3.4 Sprint Pipeline

Agile usually requires a single active "Sprint", delivered immediately to the customer, so only two builds are ever of interest:

- Build N: in operation with the customer; used to report defects.

- Build N+1: the current development sprint.

This assumes the customer is always able to accept delivery of the product and use it immediately. This isn't realistic for high-integrity projects. Some customers have their own acceptance process, and regulators may have to assess the system before deployment. These processes can be orders-of-magnitude slower than a typical Agile tempo.

iFACTS uses a deeper pipeline and multiple iteration rates, with at least four builds in the pipeline:

- Build N: in operation with the customer.
- Build N+1: undergoing customer acceptance. This process is subject to regulatory requirements, and so can take months.
- Build N+2: in development and test.
- Build N+3: undergoing requirements and formal specification.

All four pipeline stages run concurrently with multiple internal iteration rates and delivery standards. The development team can deliver to our test team several times a day. A rapid build can be delivered to the customer (in, say, 24 hours), but comes with limitations on its assurance package and allowed use: it isn't intended for operational use, but for feedback from the customer on a new feature. A full build (perhaps once every 6 months) has a complete assurance package, including a safety case, and is designed for eventual operation. The trick is to make the iteration rates *harmonic*, both with each other and with the customer and regulator's ability to accept and deploy releases.

3.5 Embedded Systems Issues

Agile presumes plentiful availability of fast testing resources to drive the development pipeline. For embedded systems, if the hardware exists, there may be just one or two target rigs that are slow, hostile to automation, and difficult to access. We have seen projects revert to 24-hour-a-day shift-working to allow access to the target hardware.

In mitigation, we reduce on-target testing with more static verification. Secondly, if we know that code is completely unambiguous, then we can justify testing on host development machines and reduce the need to repeat the test runs on target. Hardware simulation can give each developer a desktop virtual target or a fast cloud for the deployment pipeline. While virtualization of popular microprocessors is common, high-fidelity simulation of a target's operating environment remains a significant challenge.

On one embedded project, all development of code, static analysis and testing is done on developers' host machines, which are plentiful, fast, and offer a friendly environment. A final re-run of the test cases is performed on the target hard-

ware with the expectation of pass-first-time, and allowing the collection of structural coverage data at the object-code level.

4 Opportunities

High-integrity practices can complement Agile. We've already mentioned the use of static verification tools. While we have a preference for developer-led, sound analysis, we recognize that some projects might find more benefit in unsound, but easier to adopt, technologies, such as bounded model checking. Computing power is readily available to make these analyses tractable at an Agile tempo.

A second opportunity comes with the realization that, if we can automate analysis, building and testing of code, when why not automate the production of other artefacts, such as synthesis of code from formal models, traceability analysis, and all the other documentation that might be required by a particular customer, regulator or standard? An example of such an "Evidence Engine" is shown in Figure 1.

5 Commercial Issues

A crucial issue: how can we adopt an Agile approach, and still estimate, bid, win, and deliver projects at a reasonable profit? Our customers' default approach is often to require a competitive bid at a fixed price, but how can this be possible in an Agile fashion if we're brave enough to admit that *we don't know everything* at the start of a project? In most of our projects, the users, procurers, and regulators are distinct groups, all of whom may have wildly different views of what "Agile" means anyway.

We have had good experience with a two-phase approach to contracting, akin to the "architect/builder" model for building a house. Phase 1 consists of the "Up-front" work—requirements, architectural design, and construction of a skeletal satisfaction argument. The "just enough" termination criteria for phase 1 are:

- Convincing evidence that the architecture will work, meet non-functional requirements, and can accommodate foreseeable change.
- An estimate of the size (and therefore cost) of the remaining work, given the currently understood scope.
- Established ground-rules for agreeing the scope, size and additional cost of change requests, and commitment to the tempo of decision making for triage of changes and defects.

Phase 2 (possibly a competitive bid) could be planned as an iterative development, using the ideas set out above. The MULTOS CA was delivered in this fashion,

with phase 1 on a time-and-materials basis, and phase 2 on a capped-price risk-sharing contract.

6 High-Integrity Deployment Pipeline

We've used the ideas in this article at Altran, but we've yet to deploy all of them at once. An idealized Agile development process would use:

- Principled requirements engineering (Jackson 2000), concentrating initially on non-functional requirements and development of architecture, specification, and associated satisfaction arguments.
- A rolling formal specification, with just enough formality to estimate the remaining work and opening development iterations.
- An evidence engine, combining static verification, continuous regression testing, automated generation of documents and assurance evidence, and a cloud of virtualized target platforms for integration and deployment testing.
- A planned, iterative development style, starting with a partial-order over system infrastructure and features that exposes potential for parallel development. Early iterations are planned in detail, while the plans for later iterations are left open to accommodate change.

7 Conclusions

Returning to the questions posed in section 1, we could summarize our findings as follows:

- *No clash*: continuous integration, verification-driven development style, continuous regression testing, and an explicitly planned iterative approach.
- *Potential clash or inappropriate*: over-dependence on test as the sole verification activity, minimizing upfront activities in the face of non-functional requirements, the incompleteness of user stories (especially for secure systems), the need to align sprints and iteration rates with customers and regulators ability to accept deliveries, and the (non-)availability of test hardware for embedded systems.
- *Agile assumptions*: customer decision making power and tempo, availability of plentiful test hardware, and commercial and contractual models needed to "procure Agile."

- *Opportunities*: Adoption of formal languages, automated synthesis and static verification as part of the deployment pipeline. Generalization of continuous integration into an "Evidence Engine."

We're deploying these ideas on further projects, and look forward to being able to report the results. We hope others will do the same.

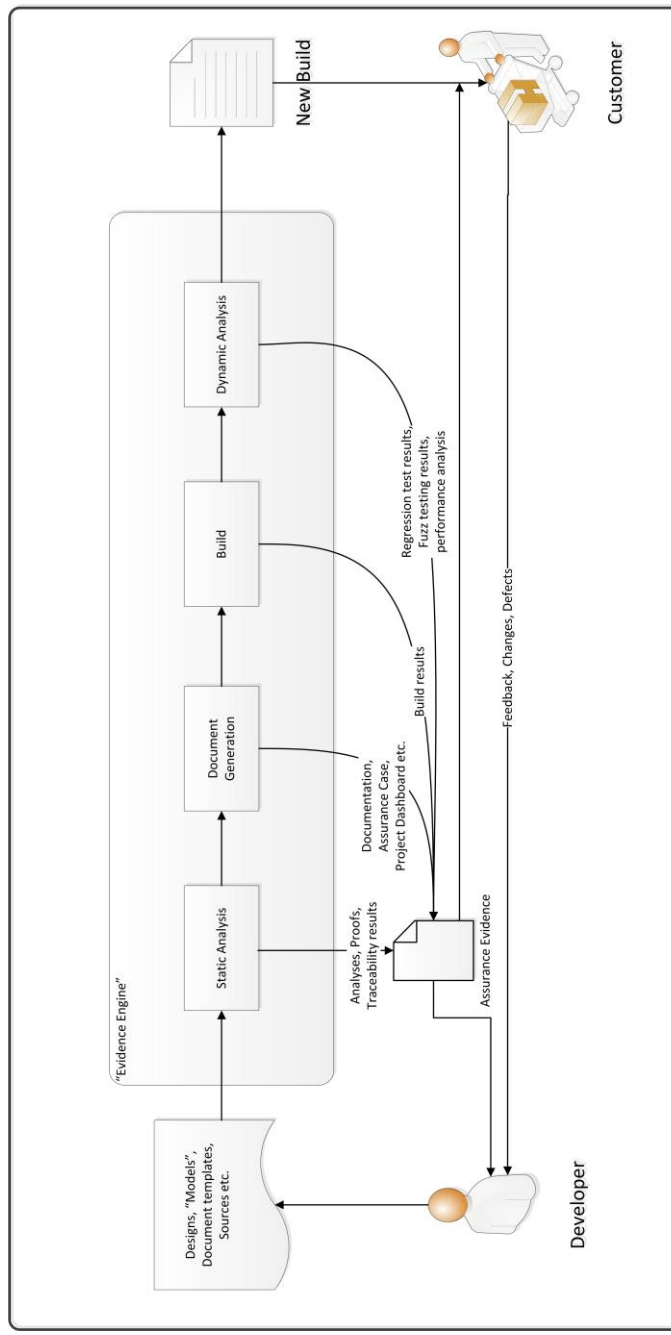

Figure 1: High-Integrity Agile Evidence Engine

Acknowledgments Thanks to Felix Redmill, Jon Davies, Mike Parsons, Harold Thimbleby and CACM's reviewers for their comments.

References

Beck K (1999) Extreme Programming Explained: Embrace Change. Addison Wesley.

Boehm B, Turner R (2003) Balancing Agility and Discipline: A Guide for the Perplexed. Addison Wesley.

Chapman R, Amey P (2003) Static Verification and Extreme Programming. Proceedings ACM SIGAda Conference.

Chapman R, Schanda F (2014) Are we there yet? 20 years of industrial theorem proving with SPARK. Proceedings of Interactive Theorem Proving 2014. Springer LNCS Vol. 8558, 17-26.

Chenu E (2009) Agility and Lean for Avionics. Thales Avionics. http://www.open-do.org/2009/05/07/avionics-agility-and-lean/ Accessed 10th March 2016.

Hall A, Chapman R (2002) Correctness by construction: building a commercial secure system. IEEE Software 19(1) 18-25.

Jackson M (2000) Problem Frames. Pearson.

Kanig J et al (2014) Explicit Assumptions – A Prenup for Marrying Static and Dynamic Program Verification. Proc Tests and Proofs 2014. Springer-Verlag LNCS, vol. 8570, pp. 142–157. DOI: 10.1007/978-3-319-09099-3_11

Larman C, Basili V (2003) Iterative and Incremental Development: A Brief History. IEEE Computer.

Meyer B (2014) Agile! The Good, the Hype and the Ugly. Springer.

Middleton P, Sutton J (2005) Lean Software Strategies. Productivity Press.

Redmill F (1997) Software Projects: Evolutionary vs Big-Bang Delivery. Wiley 1997. http://www.safetycritical.info/library/NFR/ Accessed 10th March 2016.

SINTEF (2015) SafeScrum website. http://www.sintef.no/safescrum Accessed 10th March 2016.