

Embedded Agile

Timo Punkka

timo.punkka@ngware.eu

Embedded Systems Conference 2010, Boston, ESC-241

***Abstract.** New product development (NPD) is getting more and more challenging. Change happens all the time in all dimensions, including own organization, technology, competition, and marketplace. Agile development is targeted at working in a turbulent environment driven by continuous learning. Originated from software industry, its applicability to embedded system development has been analyzed over the years. In this paper, I present some observations on implications of embedded system development to agile development. I introduce findings on frequent releasing, automated testing, co-design including non-SW development and quality systems like ISO9001.*

Introduction

“As more devices become automated and consumers acquire more such devices, the volume of embedded software is increasing at 10 to 20 percent per year depending on the domain. Embedded microprocessors account for more than 98 percent of all produced microprocessors, thus vastly surpassing computing power in the IT industry.” (Ebert and Jones, 2009) Software is everywhere and it is more complex. This means we need to write more software. Ever-accelerating rhythm of everyday life is affecting R&D. Life-cycle of electronics is getting shorter and shorter resulting in need to write the software faster. Top quality, shorter time-to-market, turbulent market place, pressure to enter new markets, and innovation, all add to modern day challenges of new product development (NPD). Traditional process models relied on assumption that it is more cost effective to first plan the project in detail and then execute according this plan. While this promise of cost effectiveness is tempting, the problem with this approach is that no one knows what is exactly needed and thus what to plan for. Furthermore the above-mentioned turbulence will most likely change or make earlier requirements obsolete. Processing requirements that will never be implemented is obviously a waste of time.

Agile methods are aiming at tackling these challenges by focusing in exploration and collaborative learning between business and engineering throughout the development. Incremental development is steered by continuous re-planning based on feedback. Changes in requirements are welcome and seen as opportunity rather than threat. Early in the millennium agile methods were considered to be suitable for small and non-critical software projects. Since then lots of experimentation has been done on large distributed programs, critical and real-time embedded systems, and even non-software development.

During my 7 years of experimenting with agile methods I have encountered several projects and teams. Projects have ranged from embedded software to co-design and hardware development. I have witnessed these methods at work with dispersed teams and distributed development. This paper documents some of the discoveries and observations I have made during this endeavour. Many of the ideas presented have crystallized while working with a high-end fire detection system development team for the last 3 years (later called the FX team). It is worth mentioning that, while some of the ideas may be a direct match to the reader, the main objective is to contribute to understanding that agile methods are not a one-size-fits-all solution. They offer an emergent, highly adaptive approach to NPD.

This paper is constructed as follows; the first chapter gives a short introduction to agile methods, embedded system constraints and how these fit together. The following chapter describes and summarizes a number of findings in this context. The final chapter presents a conclusion and discussion.

Agile development – a brief intro

In 2001 a group of software professionals got together to discuss about methods to develop software. They wrote down their shared values in agile manifesto¹:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following the plan

That is, while there is value in the items on the right, we value the items on the left more.

They further defined 12 principles to clarify the values present on cover page:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile development shares the values and principles (Abrahamson et al., 2003), but there is still room for a variety of different agile methods and frameworks. Out of them Scrum (Schwaber and Beedle, 2002 and Schwaber, 2004) and Extreme Programming (Beck, 1999 and Beck, 2004) are the most adapted in the industry. Scrum is a project management framework illustrated in Fig. 1. In Scrum work is broken down to fit short increments called Sprints. The work to be done in given Sprint is planned in a Sprint Planning Meeting. In this meeting work is pulled from a Product Backlog, and moved into team's Sprint Backlog. The work planned for a Sprint is managed by the Scrum team. A Scrum team consists of 10 or less cross-disciplined members. Ideally the team has all the skills necessary to complete the Sprint. During the Sprint, the team has a Daily Scrum Meeting to synchronize the information. The Product Owner is responsible for prioritizing the Product Backlog containing all the work to be done in a project. The Scrum Master is responsible for keeping the process fit and coaching the team into continuous improvement. At the end of the Sprint, a Sprint Review meeting is held to demonstrate the team's achievement to all stakeholders for gathering feedback. Between Sprints, the team holds a Retrospective meeting to gather improvement ideas. Scrum does not give guidance on engineering practices. For this reason teams often supplement it with practices from other methods, like Extreme Programming. (Schwaber and Beedle, 2002) express this as "if practices were candy the Scrum is the wrapping paper for candy".

¹ <http://www.agilemanifesto.org>

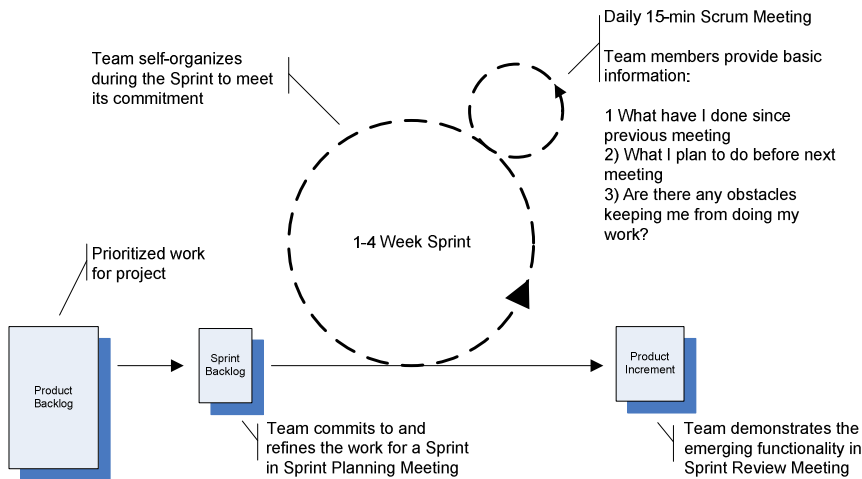


Fig. 1 Scrum Framework (Schwaber and Beedle, 2002 and Schwaber, 2004).

Unique characteristics of real-time and embedded systems

There are several concepts that are typical to real-time embedded systems. Following list is given by Bruce Douglass (Douglass, 1999):

- Timeliness
- Responsiveness
- Concurrency
- Predictability
- Correctness and robustness
- Distributed systems
- Fault tolerance and safety
- Dealing with resource-limited target environments
- Low-level hardware interfacing
- Real-time operating systems

This list is by far not complete. We can say that real-time embedded system development has several unique characteristics creating a need for creative adaptation of agile methods. Often developers do not have formal education on computer science making the situation further challenging.

Embedded Agile

Planning is called for when experimenting is expensive. Compilation power enables rapid incremental development in software development. The cost of experimenting is going down also for electronics and mechanics development. New technologies, such as rapid prototyping, FPGA, and 3D printers, enable much faster, and cheaper, cycles than previously. We can afford to fail a few times before getting it right. Stefan Thomke calls this an era of “enlightened experimentation” (Thomke, 2001). This new possibility to experiment opens the window for agile methods in embedded system development even beyond software.

The suitability of agile methods to embedded system development has been studied since the very early days of agile development. James Grenning, one of the authors of agile manifesto, concluded in 2002 that embedded systems can benefit from XP practices, especially from its strong focus on automated testing. One of the clear benefits he identified was the ability to progress with software before the hardware is available by using tests to validate the work (Grenning, 2002, Grenning 2004a and Grenning, 2004b). A similar conclusion was drawn based on a 12-month empirical study by (Ronkainen and Abrahamson, 2003). More recently multiple experience reports from industry have presented concrete

evidence on benefit of agile methods and agile testing techniques in embedded software development (Van Schooenderwoert, 2006, Karlesky, Bereza and Erickson, 2006, Fletcher, et.al. 2007 and Cordeiro, et.al, 2008). (Srinivasan, Dobrin and Lundqvist, 2009) reviewed the current literature and conducted interviews in three aerospace and defense leaders and proposed that agile methods provide means to mitigate the impact of emergent changes between systems and embedded software.

Theoretical analyses on applicability of agile methods to development outside embedded software have also been frequently presented (Smith, 2007, Highsmith, 2009 and Reinertson, 2009). However, empirical evidence from real-world industry setting is still scarce. As (Smith, 2007) states in his introduction chapter in Flexible Product Development, “However, this presents a paradox for a new field with only limited experience to present in demonstrating how the techniques apply to development projects”. He reminds us that if there was a plenty of experience data, the idea wouldn’t be new.

Findings

This chapter presents findings from number of agile embedded system development projects.

Table I Summary of findings		
Finding	Description	Recommendation
Frequent Release (Cadence)	Releases can be done on fixed dates by adjusting the scope. In practice having frequent production releases means planning with multiple time-spans.	By keeping the date and enabling full visibility to scope changes, stakeholders have the information to better communicate with customers. Working together with test agencies, considering test schedule in release planning and using disciplined software configuration management help in dealing with long approval processes. Stakeholders need to be coached in categorizing the arriving work. Very little of such work really needs immediate response.
Product team	Instead of each developer being responsible and having knowledge on a narrow area (component), everyone in a product team should have knowledge on the whole system.	Daily Scrum meetings and Continuous Integration help to avoid “stepping on each other’s toes”. At first this seems like slowing the team down as the knowledge needs to be transferred, but this can be accelerated with coaching and lowering the bar for asking help.
Automated Testing Using Dual Targeting	Fast test cycles are essential to agile development. Dual targeting means being able to run the application and parts of the application in isolation in development environment. This leads to faster and more flexible automated testing throughout the lifecycle.	Start dual targeting immediately and keep the code compiling all the time for both or multiple environments. Testing in target environment is also needed.
Continuous Integration	A Continuous Integration (CI) server monitors the version control repository and triggers a build when changes are detected. A build includes phases for testing against integration defects and reporting to developers.	You can start CI with just a version control and an automated build. Grow your definition of done continuously. Scripting languages, such as Python and Ruby, can be used as ‘super glue’ between different tools and the CI server.

Co-design and non-SW agile development	Computing power, reduced price of tools and advancement in prototyping technologies have dramatically lowered the cost of experimenting in non-SW development as well.	For ease of planning and synchronization keep hardware and software teams in the same rhythm. This also creates a sense of common direction. When HW and SW developers are in the same team, remember that the other discipline cannot understand the technical details of the other and adjust meetings accordingly.
ISO9001	In an overly simplified model ISO9001 requires documented quality assurance and product realization process or processes. Conformance needs to be proven by records. In many industries ISO9001 certificate is a prerequisite for business so this extra recording can be seen as unavoidable cost of doing business.	Describe your incremental agile process model in your project plan. Keep records of your incremental planning and validation techniques. For example, records of the Product Backlog and automated test report after each Sprint are suitable for demonstrating continuous re-planning and validation respectively.
Customer is a role	In agile development the prioritization should be customer driven. However, in embedded system development a number of variables affect prioritization. Customer is rather a role played by different people.	Having a single person as the Product Owner (PO) helps the team to focus. The PO should however facilitate a team of different domain experts to form her opinion.

Frequent Release (Cadence)

- "We need a date."*
- "We can't release after every Sprint. The test agency takes a year."*
- "What about the urgent stuff?"*

Despite the fact that this is what we most often hear, I have noticed that it is not necessarily the speed that business looks for in R&D. Far more important is predictability. To deliver when you say you will deliver. According to my experience scope reductions are very well tolerated when this is openly communicated in advance and as long as you still deliver the most valuable features. As a product manager explained:

"Visibility has increased. It is easier to collaborate with customers about priorities and schedules."

When you establish a steady cadence you will save lot of time from crisis meetings called on for delays. This in turn lets you focus on progressing instead of explaining. Fig. 2 illustrates the cadence for FX team. There are several overlapping planning horizons. Most of the time there is a longer term effort going on which may stretch over several release cycles. However, the team delivers something to production three times a year on fixed dates on January, May and September. On one of these dates it will be the results from major project effort, perhaps only after an approval from a test agency. More frequent releasing is possible but at the moment not seen practical by partners.

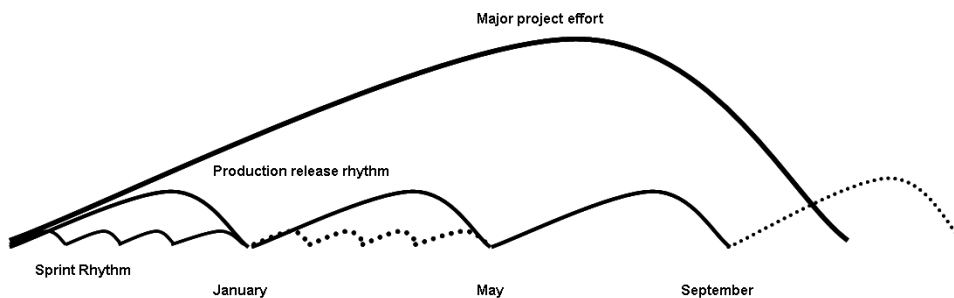


Fig. 2 Overlapping planning horizons.

All these cycles are made visible by agile practices. Inside a Sprint, the team manages the tasks using a Scrum board and discusses progress in Daily Scrum meetings. The Product Backlog is on another board and is reworked in a weekly meeting by the Product Owner and stakeholders, mainly product managers. Weekly meetings also review the portfolio on third board. Prioritizing of the current release and upcoming projects is visible all the time. The status of development and prioritization is reviewed monthly by the top management. Although cadence sounds like a wonderful idea, there are a few things working against us, like official testing agencies and urgent work. I believe every domain has similar impediments.

The fire detection market is strictly controlled by authority. The official full blown test cycle can take anything from a month to a year costing tens if not hundreds of thousands of Euros. How can one be agile and flexible in such an environment? FX team uses *mainline* and *branch before releasing* software configuration management patterns (Berczuk and Appleton, 2003). All the active development happens in single mainline, or trunk. When closing to a date of release, they create a release branch. Main development continues in the trunk. The final test is expected to find nothing due the continuous testing during the development. If defects are found in the release branch, they are fixed there and then merged back to the trunk as well. When a next release is to be done the previous release branch becomes a dead end and a new release branch is created from the trunk. The idea is illustrated in Fig. 3. New release branches should not be created from the previous ones and there should not be any longer development in a branch. Long-living branches will be difficult to merge back to the mainline. Even while massive merges should be rare, invest on high quality 3-way diff/merge tools.

A recent analysis on agile methods and embedded systems development (Srinivasan, Dobrin and Lundqvist, 2009) stated “When it comes to mission-critical systems in particular, the standards highlight the importance of creating the necessary artifacts to enhance trust in the implemented system, but do not necessarily specify the process to be followed to generate the necessary artifacts.” There is room for making working with test agencies more flexible. For example, we can build more partnership-like relations with the agencies and establish trust with alternative means instead of traditional written documentation. The automated test suite, for example, offers possibilities for novel ways of communication.

Many test agencies allow the customer to define levels of change, for example in terms of levels 1-3. The lowest are minor modifications that do not need a formal test period. Considering this in release planning allows you to ship the version for testing earlier. You then finish these minor modifications, or requirement areas that are not under the test procedure, during the testing process. If the certificate from the test agency is not available at the fixed release date, the functionality requiring approval is removed, and other functionality is still released to production. There are a few techniques to remove the functionality, which has not yet been given an approval, from production release;

- 1) The functionality is blocked in the configuration software
- 2) The functionality is blocked with a conditional compilation
- 3) The release is built from an earlier branch (the last option)

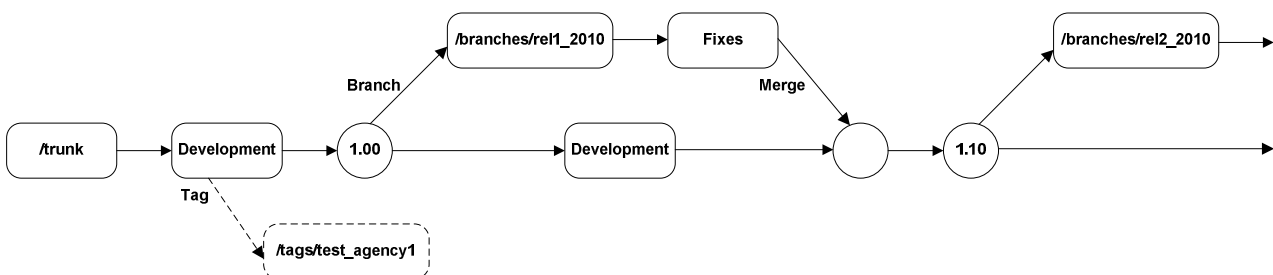


Fig. 3 Branch before releasing –software configuration management pattern.

Some literature recommends dedicated maintenance teams and that this responsibility should be rotated. It may be needed for very large programs, but I have never experimented with dedicated maintenance teams or team members. This work has proven to be fairly constant over a longer period of time. For this reason, I don't plan for maintenance work during the Sprint Planning. I would do that only if I needed to illustrate the amount of maintenance work to someone. This is actually something that can help new teams and their managers to see the amount of interruptions. To

make this visible, post a red task card on Scrum board for each interrupting task during the Sprint. In the Retrospective meeting ask for each interruption if it really needed immediate attention compared to planned work for the Sprint. There are, of course, issues that need immediate attention, but a huge improvement can be achieved by coaching people to categorize their issues:

1. Things to deal within this minute. This interrupts the development. This can be a problem in production or defect causing danger in the field.
2. Things to deal in the Daily Scrum meeting. The impact of the interruption is smaller as the team is planning for the next day.
3. Things that can be processed through the Product Backlog during the current release.
4. Ideas to be prioritized through the Roadmap Backlog.

The Scrum Master should keep an eye on if someone is taking the responsibility of maintenance alone and coach the team to share these skills as well.

Product team

-“But we will step each others toes.”

Often development responsibility of a large system is divided based on its components. A component can be a software component or a physical component like a single printed circuit board (PCB) in modular system; “Mark is the only one who knows about the serial communication driver and protocol”. The standard justification for this is that it is more efficient for one person to be specialized on narrow area and hold the complete responsibility for this. While nice in theory, this causes several problems in practice. Most of the end-user visible features need development of many, if not all, components. So firstly, instead of enabling one person to focus on single thing, she is in fact forced to work in a virtual multi-project environment serving all the other developers’ requests. This makes the development highly unpredictable. Secondly, this practice makes company vulnerable for several risks. The person with specialized skill set becomes easily a bottleneck creating a queue of unfinished work and thus lengthening the feedback cycles and delaying learning. The person may also leave the company due different reasons; moving to another company or winning the lottery. It is also easier to get help and to create innovative solutions when you have peers that can discuss about the area you are working on. A developer in FX team explained the change he had felt after moving to work on a single project as team:

“...it is easier to ask help and collaborate as everyone is sharing the same project’s responsibilities compared to earlier when everyone was busy with their own responsibilities.”

The FX team was initially a group of specialists, each with his responsibility. There was not a strict division of responsibilities, but even this led to a situation where 5-7 projects were carried out at the same time. This causes the third problem. Specialized developers force the portfolio management to be driven by skills, not by the company vision and strategy. The company develops what it can and not what it wants. One of the projects already started when the FX team moved into being a product team, was later removed from the portfolio as not needed by marketing. This illustrates the fact that component-based specialists lead to a less than optimal portfolio management. It is not necessary to have pure equal generalists in your team but rather “T-shape” developers with knowledge on the system as a whole, but also deep expertise on some areas. In this situation losing a developer may slow the team down a bit, but progress will never again be blocked by the absence of a single person. At first, breaking the knowledge silos means that developers start to work on features that they have very little knowledge on. As a facilitator, you need to encourage this by lowering the bar for asking help. Remember that all experts are playing in your team towards a shared goal. This also reduces the risk of missing fixed release date due to the absence of a person. This idea is presented as feature teams instead of component teams by Craig Larman and Bas Vodde (2010). When moving to product team, certain practices are necessary to help to avoid stepping on each others’ toes. For example, the Daily Scrum meeting facilitates information synchronization between team members and the Continuous Integration helps in verification against integration defects.

Automated Testing Using Dual Targeting

*-"We don't have hardware to run the code in."
-"Every part of the code is hardware dependent."*

Dual targeting is a technique to separate the target hardware dependent parts, like hardware register manipulation and operating system (OS) calls, from the hardware independent application. By doing this, you are able to compile the majority of your source code to be run on the development environment as well. This results in an ability to execute much faster code-test cycles than would be possible in the target environment. In addition, you invest on flexibility on hardware and OS changes during the development and even after the first launch. There is nothing dramatically new to this. Mainstream programming uses the same kind of techniques for automated testing when tests are needed to be isolated from the slow or otherwise difficult-to-involve parts of the system, such as database. See, for example, hexagonal (later ports and adapters)¹ and onion² architecture patterns.

There are several techniques to do this in practice (Grenning, 2010). Simplest is likely to be the conditional compilation, but this leads to a mess rather sooner than later. Most pragmatic for embedded C applications is the link time polymorphism (Koss and Langr, 2002). This simply means that a different implementation of the hardware dependant interface gets linked for each target. In practice it may be a combination of several techniques.

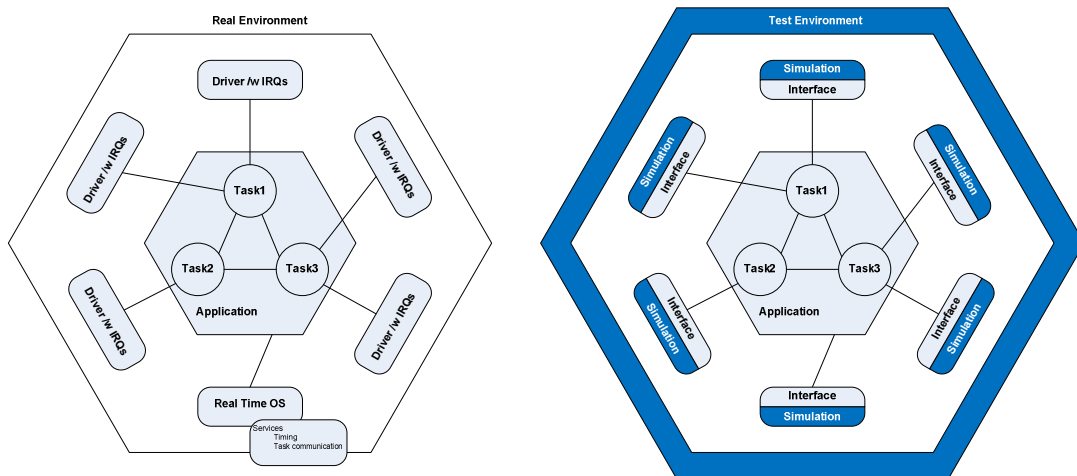


Fig. 4 The principle of dual targeting.

The FX team compiles software for different processor cards for two platforms throughout the lifecycle. The code is cross-compiled for the target hardware, but also for the PC simulation by replacing all hardware dependent parts with the simulation implementation using the same interfaces (Fig. 4). The simulation is run on platform called E-SIM. This tool is now discontinued, but Mentor Graphic's EDGE SimTest³ is the continuum of this tool. E-SIM comes with a Python API and a simple acceptance test framework is developed in Python enabling test sequences to be written in a simple text format. Below is a conceptual example of a test script. The script launches a simulation for the main CPU card (MC) and for one fire detection loop control card (SLC). One fire detector (address 001 in loop 1) is set to detect fire. The system is expected to indicate this with blinking fire led. Next, the detector is deactivated and the fire led is expected to switch off. The two boards are running as separate processes in a PC and the serial communication between them is simulated via E-SIM's Communication Manager running as a Windows service. The test execution process is illustrated in Fig. 5.

¹ <http://alistair.cockburn.us/Hexagonal+architecture>

² <http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>

³ http://www.mentor.com/products/embedded_software/edge-dev-suite/simtest/

```

configuration N1_MC,N1_SLCl

detectFire SLCl, Set, 1.001
expectLedTo Blink, FireLed

detectFire SLCl, Clear, 1.001
expectLedTo ContOff, FireLed

```

DetectFire and expectLedTo are just simple examples of keywords available. In general this approach is called keyword driven development. Similar open source test frameworks are Fit/Fitnesse/Slim¹, Robot Framework², Systir³, and Exactor⁴. All these can be adapted to work in embedded software development using the dual targeting technique.

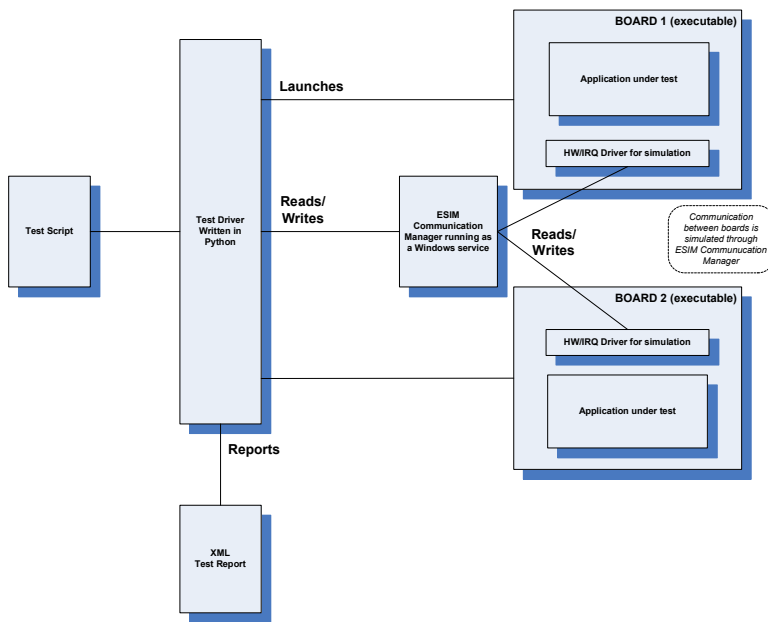


Fig. 5 Automated test execution process.

Automated testing is needed at different levels. We need acceptance tests described above to demonstrate the end features, but it is impossible to get a decent coverage with these tests. Unit tests provide a means to achieve high test coverage and to avoid those small mistakes developers do when they translate their ideas into code. This practice further helps in locating the cause of failure between software and hardware (Van Schooenderwoert, 2004). Unit tests provide a good tool for creating tests for cases like unexpected interrupt sequence caused by glitches or communication errors caused by interference. Traditionally these are just expected to work or, at best, tested once with some special code or even hardware. The same dual targeting technique enables us to write isolated unit tests as well, but beginning to do so with a large base of legacy code is not so easy. There are a few things that have been found helpful with the FX team;

- 1) automated mocks using CMock⁵
- 2) RTOS task handlers as pinch points
- 3) small group workshops

CMock is a mock generator for C. It reads the header file for a module and generates a corresponding mock implementation. In a test case, we can set expectations for interaction and validate these expectations at the end. Generated mocks help in not having to write test fakes by hand.

¹ <http://fitnesse.org/>

² <http://code.google.com/p/robotframework/>

³ <http://rubyforge.org/projects/systir>

⁴ <http://exactor.sourceforge.net/>

⁵ <http://sourceforge.net/apps/trac/cmock/wiki>

Michael Feathers (2004) defines a *pinch point* as a function or a small collection of functions that you can write tests against and cover changes in many more functions. The FX team achieved coverage quickly by building RTOS tasks as testable units and injecting stimulus to task's event handler. Behavior was then sensed using different techniques, like;

1. automated verification through generated mocks
2. using getters of utilities, like linked lists
3. making internal status visible to tests by few nasty little tricks like #define STATIC
4. temporarily using sensing variables (Feathers, 2004)

In the beginning some developers get interested in unit testing before the others. Arranging workshops among these early adopters has been seen helpful. At first, the team has gaps in experience and understanding in the team. It helps to get the people at the same level to discuss the problems and key findings. Results from these workshops can then be shared with the whole team as lessons learned, further easing the adoption by the rest of the team.

Naturally we still need to include testing in target environment as explained in embedded TDD cycle (Grenning, 2004b). When these tests find something we can write new tests to prove the HW driver incapability (when test fails) and use the same tests to validate the fix (when test passes). A positive side effect is the continuous expanding of the regression test suite.

Continuous Integration

*"I don't know how long it takes to release."
"It works on my computer."*

Continuous Integration (CI) is commonly understood as a system monitoring the team's version control repository. Developers should check-in, or commit, their changes at least daily. When a change is detected, the CI server updates its own workspace and triggers an automated build. The build includes automated tests to detect integration defects soon after they are created, thus reducing the effort needed to fix them. Typically the CI server publishes the results, for example, on a big screen in team's room or by sending a report on email.

There are many open source continuous integration servers available, for example, CruiseControl¹ and Hudson². Both of them have a wide support for different tools and they are both easy to extend. Modifiability and extensibility are important characteristics in selecting the server for embedded projects as build often includes several different tools, many of them custom-made for the given project/product. Phases in FX team's current CI schedule are:

- Building different processor card executables for simulation on a PC (MS Visual Studio)
- Building for the target and making the binaries accessible for the QA (gcc and IAR cross-compilers)
- Running the acceptance test scripts with Python test driver
- Building and running unit tests (rake and gcc)
- Creating quality metrics (CCCC³ and CCFinder⁴)

The build script also parses the results from each phase and translates this into XML to be consumed by CruiseControl. The results are distributed via email. CruiseControl relies heavily on XML for reporting the build results. Scripting languages, such as Python and Ruby, can be used as super glue between many different tools in the embedded tool chain and CruiseControl. You should not delay implementing CI. You can start simple. All you need in the beginning is a version control system and an automated build. The automated build means that you can do something concrete via a command line interface, for example, executing a Makefile. Even from this simple task, you get immediate payoff by removing the risk of having local changes in one developers working copy – to avoid the so-called “works on my computer” –situation. After you get this running you can add tasks to the build loop as you continuously grow your Definition of Done⁵.

¹ <http://cruisecontrol.sourceforge.net/>

² <http://hudson-ci.org/>

³ <http://sourceforge.net/projects/cccc/>

⁴ <http://www.ccfinder.net/>

⁵ Common understanding between development and business what does it mean for a feature to be “done” in a Sprint.

Co-Design and Non-SW Agile Development

-*"Hardware is always late."*

-*"Software always under performs."*

It has earlier been laborious to make PCB layout changes, or even schematic changes. When we go back in time, software changes were expensive. It made sense to try to get it right the first time. However, development in tools and increase in computing power have changed the rules of the game. Developers from different disciplines need to change their mindset from considering change as negative rework towards considering it as positive opportunity of learning.

A traditional embedded system process has an up-front activity called system design. The objective of the phase is to decide the partitioning and interfaces between software and hardware. There are often a myriad of parameters in the equation; the single cost of software development vs. re-occurring cost of HW components, performance issues, etc. It seems that change in any parameter affects every tradeoff creating a multi-parameter, multi-loop feedback system to be solved. Like this isn't enough, we do not know everything about the system we are trying to optimize. In such an environment it is proposed to be more effective to move rapid prototyping to early phases of development for solution finding (Clay and Smith, 2000). In embedded system design this means hardware, software and requirements emerging concurrently.

Today it is even possible to iteratively develop a physical prototype inside a Sprint. Fig. 6 shows how a RF controlled light dimmer evolved inside a 4-week Sprint. The development team consisted of firmware, electronics, PCB layout and plastic designers. The prototype included 3 PCB's; a power, a control and an UI including the RF transmitter. The first iteration had only the power PCB ready and the other 2 were replaced with common prototyping boards. On second iteration all boards were available and at the end of the Sprint rapid prototypes for plastic parts arrived. In the Sprint review prototypes gave a real feel of how the product would look like if this concept was to be selected. Firmware was developed for a simple on/off functionality. To be able to do this you need trusted partnership with prototype manufacturing. In this case the prototyping shops agreed to deliver the next day after receiving final drawings on earlier-agreed time.



Fig. 6 Pictures of an iteratively evolving physical prototype inside a Sprint.

The number of cycles in hardware development depends on cost of the cycle, the cycle time and the extent of unknown. Exploring with physical prototypes, even if you try to avoid rework, introduce a repetitive cost called a cycle cost (Smith, 2007). This accumulates from the extra work when developers process drawings for prototype manufacturer, material and labour (assembling and testing) from manufacturing, shipping, etc. Often teams can find ways to minimize these costs for example by using computer simulation and automating repetitive tasks.

The FX HW team has a similar Product Backlog as the SW team. The teams are in the same Sprint rhythm and work is synchronized between the Sprints. The fire panel boards have a longer cycle time than a Sprint and the development progresses as a more or less sequential process, but there is benefit in having frequent synchronization points between the backlogs of the HW and SW teams. Even having the hardware and software people in the same team steer towards a shared goal. In such a case you need to be more disciplined with the level of information you share on the Daily Scrum meeting. Each discipline should dig into more detail in smaller groups afterwards. Van Schooenderwoert and Morsicato present how hardware evolved in steps in a project they were involved in (Van Schooenderwoert and Morsicato, 2004).

They state that “this dovetails nicely with iterative software development”. (Highsmith, 2002) concludes that “In reality, these separations [of HW and SW teams] cause continuous conflicts as crunch time approaches when hardware and software must be integrated.” Longer cycle hardware development does not hinder iterative and incremental, agile, development, but demands it.

ISO9001

“Plan to re-plan and then show that you did so.”

At least in Europe, having the ISO9001 certificate is a prerequisite for being in the business. Papers about fitting agile methods into ISO9001 have been presented earlier (for example Namioka and Bran, 2004 and Stålhane and Hanssen, 2008). ISO9001 requirements, in overly simplified form, state that a company needs to have defined quality assurance and product realization processes and records to prove that these are followed. FX team’s processes were ISO9001 certified before introduction of agile methods. The process followed a Stage-Gate (Cooper, 2001) process model. After introducing agile methods, the process needed to be modified, and it was successfully assessed in 2009. The new definition was aimed to allow high flexibility to situation of a given project. A two-page definition of incremental development was added to the process definition among with one-page guidelines for combining stages in Stage-Gate model. For example the releases for software development have a project plan which states the mission for the release, that the detailed planning is to be done iteratively, and that for this reason all phases are combined into one: Development. A project plan fits on single page and is companied with another page recording the reasoning for the prioritization. For larger effort, a simple few-page product specification is written defining the high level constraints for system.

A template for a project plan:

- Project details; the name, number, product manager, Product Owner and the team.
- Tradeoff matrix; what is to be sacrificed
- Exploration factor; a number illustrating the amount of unknown based on two parameters, requirements and technology
- Objectives; the mission statement, schedule, cost, and production estimates
- Other issues; for example that all phases are combined, the specification is replaced with a backlog and the development will be incremental
- Marketing argumentation; the reasoning for the prioritization

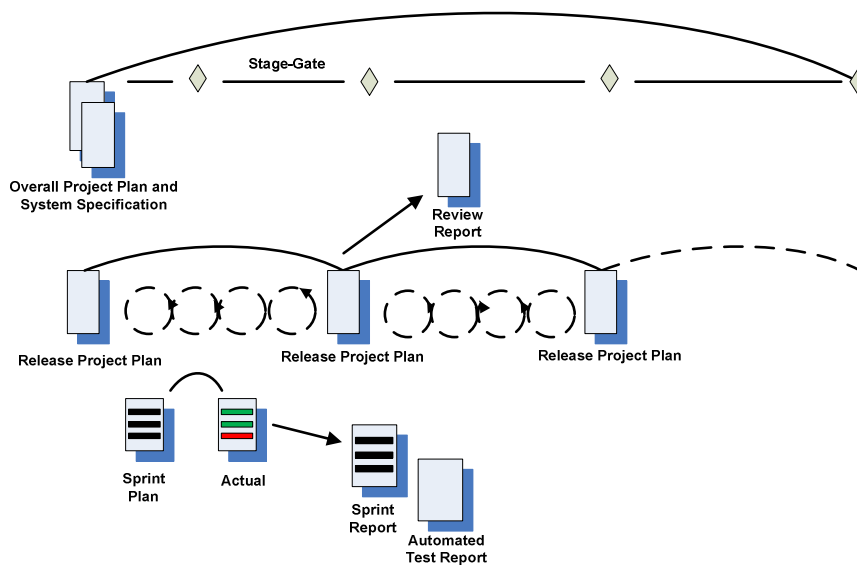


Fig. 7 Records from the process for ISO9001.

We keep records of the Scrum board as snapshots in electronic form after each Sprint Planning and then again at the end of the Sprint (Fig. 7). A single-page Sprint Report is written after each Sprint Review. It shows the planned versus actual achievement and a velocity trend for the hardware and software teams. It also has a list of participants, a short summary of what was demonstrated, and the feedback gathered during the Sprint Review. A similar approach can be adapted for validation process as well. A high-level test plan can be created, or integrated into project plan, stating that more detailed test cases will be incrementally developed. A continuously growing automated test suite produces validation reports at the end of the each Sprint. These records together form the audit trail as the artifacts are developed in parallel with the development in short iterations. This approach was appreciated by the auditor for the simple visual trail it provided.

When there is major hardware development involved, the hardware is run with a high level Stage-Gate process implementing all gates. This is seen as helping in larger purchase decisions and moving towards maturity for mass production. In parallel we still run development using the release and Sprint planning. In general gates have less meaning as you have much more frequent decision-making points between Sprints.

Customer is a role (Product Owner team)

-“Who is the onsite customer?”

The Product Owner is a single person. The person needs to be fluent in two languages; technical and business as well as have excellent social and negotiation skills. It is fair to say that it is difficult to find a person with these skills. In an embedded project the real customer of embedded software varies during the life-cycle. Early on, it may be the hardware team or team members who need some driver software to verify component choices. The Product Owner is responsible for prioritizing the Product Backlog. There are however a myriad of parameters affecting the priority, for example;

- Marketing
- Other team(s)
- Risk reduction and learning
- Technology
- Production
- Purchase
- Standards
- Competition
- Next production release
- System backward compatibility
- Training programs
- Operations and maintenance
- Ease of commissioning
- etc.

Having knowledge on all these aspects (and there are more) by a single person is virtually impossible, but having a single person responsible helps the team to stay focused. This is why the Product Owner needs to have a trusted team of domain experts working with her. Moreover she needs to appreciate all the different stakeholders and be able to prioritize, and re-prioritize, between these.

Discussion and Conclusion

The size and complexity of embedded software and systems keeps growing. Time from idea to cash needs shrinking all the time. Greenfield projects are rare and enhancement of existing systems is typical. Change is everyday reality and continuous learning is needed. Agile methods aim at tackling these challenges. In this paper I have scratched the surface on what I have learned over the years about adopting agile methods to embedded software and system development. It is clear that agile methods are applicable to embedded system development. At the end of the day challenges are not so much different from any other kind of development work. Inspect and adapt –cycles naturally help you in refining the

method for your given context.

Agile methods emphasize the frequent delivery of valuable development results to production. At first glimpse, it seems that embedded system development has several unique characteristics playing against feasibility of these methods. In this paper, I have given a high level introduction on how a few of these issues can be overcome. Based on these few insights only we can see agile practices and techniques both support and depend on each other. Frequent delivery is not something that you can just decide to do. Practices that support this goal include the product team, automated regression testing, continuous integration, and iterative synchronized co-design. On the other hand agile transition does not happen in isolation. When R&D changes the way they work it has implications on several boundaries. In this paper, changes in a few boundaries were discussed including marketing, top management reporting, and ISO 9001. It is still a fairly common misconception that agile is only affecting engineering. Introducing the agile toolbox to engineering is only the first step. Changes in other functions are also needed in order to harness the full potential of agile R&D. The organization should focus on optimizing the performance of the whole system. There is a lot of work going on around agile enterprise and this applies to companies involved with embedded system development as well.

There is no end state in agile transition. It is rather the *state of mind to challenge everything* that needs to be achieved throughout an organization to truly “be agile”.

Reference

- (Abrahamson et al., 2003) Abrahamson Pekka, Warsta Juhani, Siponen Mikko T. and Ronkainen Jussi, New Directions on Agile Methods: A Comparative Analysis, Proceedings of the 25th International Conference on Software Engineering (ICSE'03), 2003.
- (Beck, 1999) Beck, Kent, Extreme Programming Explained: Embrace Change, Addison-Wesley 1999.
- (Beck, 2004) Beck, Kent, Extreme Programming Explained: Embrace Change, 2nd Edition, Addison-Wesley Professional, 2004.
- (Berczuk and Appleton, 2003) Berczuk, Stephen P. and Appleton, Brad, Software Configuration Management Patterns – Effective Teamwork, Practical Integration, Addison-Wesley, 2003.
- (Clay and Smith, 2000) G. Thomas Clay and Preston G. Smith, Rapid Prototyping Accelerates the Design Process, Machine Design, pp.166-171, March 9, 2000.
- (Cooper, 2001) Dr. Cooper, Robert G., Winning at New Products: Accelerating the Process from Idea to Launch, 3rd edition, Basic Books, 2001.
- (Cordeiro, et.al., 2008) Cordeiro, Lucas, Mar, Carlos, Valentin, Eduardo, Cruz, Fabiano, Patrick, Daniel, Barreto, Raimundo and Lucena, Vicente, An agile development methodology applied to embedded control software under stringent hardware constraints, SIGSOFT Softw. Eng. Notes 33, 2008.
- (Douglas, 1999) Douglas, Bruce Powell, Doing Hard time, Addison-Wesley 1999.
- (Ebert and Jones, 2009) Ebert, Christof and Jones, Capers, Embedded Software: Facts, Figures, and Future, IEEE Computer, vol. 42, no. 4, pp.42-52, 2009.
- (Feathers, 2004) Feathers, Michael, Working Effectively With Legacy Code, Prentice Hall, 2004.
- (Fletcher, et.al., 2007) Fletcher, Matt, Bereza, William, Karlesky, Mike and Williams, Creg, Evolving into Embedded Development, Proceedings of AGILE 2007 Conference, IEEE, 2007.
- (Grenning, 2002) Grenning, James W., Extreme Programming and Embedded Software Development, Embedded Systems Conference, 2002. Available at <http://renaissancesoftware.net/papers>
- (Grenning, 2004a) Grenning, James W., Progress Before Hardware, 2004. Available at <http://renaissancesoftware.net/papers>
- (Grenning, 2004b) Grenning, James, W., Embedded Test Driven Development Cycle, 2004. Available at <http://renaissancesoftware.net/papers>
- (Grenning, 2010) Grenning, James, W., Test Driven Development for Embedded C, The Pragmatic Programmers LLC, 2010.

Forthcoming, available in beta at the time of writing at <http://www.pragprog.com/titles/jgade/test-driven-development-for-embedded-c>

(Highsmith, 2002) Highsmith, Jim, Product Development and Agile Methods, The Cutter Edge, 2002.
Available at <http://www.cutter.com/research/2002/edge020528.html>

(Highsmith, 2009) Highsmith, Jim, Agile Project Management: Creating Innovative Products, 2nd edition, Addison Wesley Professional, 2009.

(Karlesky, Bereza and Erickson, 2006) Karlesky, Mike, Bereza, William, and Erickson, Carl, Effective Test Driven Development for Embedded Software, Proceedings IEEE Electro/Information Technology Conference, 2006.

(Koss and Langr, 2002) Dr. Koss, Robert and Langr, Jeff, Test Driven Development in C, C/C++ Users Journal, October 2002.
Available at <http://www.objectmentor.com/resources/articles/TDDinC.pdf>

(Larman and Vodde, 2010). Larman Craig and Vodde Bas, Practices for Scaling Lean & Agile Development, Addison-Wesley, 2010.

(Namioka and Bran, 2004), Namioka, Aki and Bran, Cary, eXtreme ISO ?!?, Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2004.

(Reinertson, 2009) Reinertson, Donald G., The Principles of Product Development Flow – Second Generation Lean Product Development, Celeritas Publishing, Redondo Beach, California, 2009.

(Ronkainen and Abrahamson, 2003), Ronkainen, Jussi and Abrahamson, Pekka, Software Development under Stringent Hardware Constraints: Do Agile Methods Have a Chance, Proceedings of the 4th international conference on Extreme Programming and agile processes in software engineering, 2003.

(Schwaber and Beedle, 2002) Schwaber, Ken, and Beedle, Mike, Agile Software Development with Scrum, Prentice Hall 2002.

(Schwaber, 2004) Schwaber, Kent, Agile Project Management with Scrum, Microsoft Press, 2004.

(Smith, 2007) Smith, Preston G., Flexible Product Development, Jossey-Bass, 2007.

(Srinivasan, Dobrin and Lundqvist, 2009), Srinivasan Jayakanth, Dobrin Radu and Lundqvist Kristina, 'State of the Art' in Using Agile Methods for Embedded Systems Development, 33rd Annual IEEE International Computer Software and Application Conference, 2009.

(Stålhane and Hanssen, 2008) Stålhane, Tor and Hanssen, Geir Kjetil, The application of ISO 9001 to agile software development, Lecture Notes in Computer Science; Vol. 5089, Proceedings of the 9th International conference in Product-Focused Software Process Improvement, 2008.

(Thomke, 2001) Thomke, Stefan, Enlightened Experimentation: The New Imperative for Innovation, Harvard Business Review, Feb 2001.

(Van Schooenderwoert and Morsicato, 2004) Van Schooenderwoert, Nancy, Mordicato, Ron, Taming the Embedded Tiger – Agile Test Techniques for Embedded Software, Proceedings of the Agile Development Conference, IEEE, 2004.

(Van Schooenderwoert, 2004) Van Schooenderwoert, Nancy, Embedded Extreme Programming: An Experience Report, Embedded Systems Conference Boston, 2004.

(Van Schooenderwoert, 2006) Van Schooenderwoert, Nancy, Embedded Agile Project by the Numbers With Newbies, Proceedings of AGILE 2006 Conference, IEEE, 2006.