Agile Methods and Firmware Development

Timo Punkka

Helsinki University of Technology, Software Business and Engineering Institute

timo.punkka@fi.schneider-electric.com http://ng-embedded.blogspot.com

Abstract— **The size and complexity of software continues to grow at a steady pace. This is also true for software embedded in our everyday electronics, which we have called simple devices. The term firmware is used to describe the low-level software in embedded systems. It may even be hard to divide firmware and actual hardware. Software development for such a target has special characteristics such as a culture of hacking, small teams and multiple hats, co-design issues, one-time designs, correctness and robustness requirements, lack of tools and unconventional customers. Software process models have been studied also in this environment to ease the pain of developing more complex systems. I introduce four currently used methods to develop firmware; build-and-fix, waterfall, ROPES and RUP SE.**

Agile methods are getting a lot of attention in the software development community at the moment. I review the agile methods which are most documented. The suitability of these to firmware development is evaluated. It is also analyzed whether firmware development could benefit from agile methods.

It is shown that agile methods are not the new cure-all solution to firmware development, but they are applicable. Their full use needs modification and innovative thinking. It is, however, shown that firmware development can surely benefit from the usage of agile methods.

Index Terms— **Agile, embedded, firmware, process model**

I. INTRODUCTION

(Gibbs, 1994) warns about Software's Chronic Crisis. As the system complexity keeps growing, the software development methods have problems to meet the challenge. To ease the pain, methods for developing software have been studied for a long time. The majority of the studies consider contract based PC software written in large software organizations.

In his article, Gibbs quoted Mary M. Shaw as "When Computers are embedded in light switches, you've got to get the software right the first time because you're not going to have a change to update it." Today microcontrollers are getting so cheap that even the simplest gizmos will have more complex software, and greater parts of hardware will be replaced by software. This lowest level of embedded software we call firmware. It may even be hard to divide it from the hardware. Is firmware development going to have a crisis of its own? To avoid this we must develop the process we use to develop firmware.

The software process is the way we produce software. It incorporates the software life-cycle model, the tools we use, and most important of all, the individuals building the software (Schach, 2004). Software life-cycle is the time from the first requirements to the retirement of the particular piece of software. There is a well known relationship between software productivity, quality and software process (Schach, 2004). However, the task of finding the right process model is far from easy. There are several reasons why software processes vary so drastically from opinion to opinion and from organization to organization. The bottom line is that no one size fits all. At the end of the day, the process appropriate to a given organization and to a given project depends on the individuals developing the software and the environment where the development takes place. In addition, firmware development places extra constraints and difficulties to development. This means also that finding an appropriate way of developing firmware is different from mainstream programming.

The conventional way of producing software is the sequential process model, like the waterfall model (Royce, 1970). The idea behind the process model is that software life-cycle phases are following each other sequentially, one after another, starting from gathering the requirements. Requirements should be iron-bound before proceeding to design and implementation. This is pure fiction today; the customer does not know what he wants at the pre-study phase of a project. Even if he has an idea at the beginning, the markets are at high risk to change during the development phase. Often these heavy process models are not followed in a small company. Instead, in an incidence of the smallest difficulty (for example a change in requirements), the process gets ditched and hacking is started again. The required documents are not updated and they fall out of synchronization with the actual development. This means that the hard work at the beginning is wasted anyway.

To tackle these difficulties, the software community has come up with a different approach in contrast to conventional heavy process models. This approach embraces the change and instead of being document-driven it is feature-driven. These so called *agile process models* are considered to be applicable for small, non-critical, software projects with vague, or rapidly changing, requirements (Beck, 2000).

I review the agile process models which are mostly written about. They are reflected to low-level embedded real-time (firmware) programming. By doing this I search for answers to two research questions:

1. Can agile process models be applied in firmware development?
2. Can firmware development benefit from applying agile process models?

The first objective is to show that agile process models can be adapted to the firmware development environment. This is to generate interest in process models, particularly agile methods, in the firmware community. The second goal is to show that firmware development can benefit from using agile methods. I will also provide another, demanding, perspective for process model developers.

I do not recommend any particular process model or process improvement plan, but give an objective and introductory view of the subject.

The theoretical part of this paper is done as a literature search. I have chosen the most documented agile methods to be reviewed. The cornerstone books on the subject are reviewed and supplemented with conference papers and major journal publications. The theoretical part is referenced to the author's personal opinions and ten years of experience in the industry.

The rest of this paper is constructed as follows: The first chapter defines firmware and its unique characteristics. The second chapter talks about the more traditional methods of firmware development. The following chapter goes through the idea of agile models and introduces the agile models that are most widely documented. The next chapter discusses the agile approach in contrast to firmware development. The final chapter wraps it all together in a summary of the paper.

II. FIRMWARE DEVELOPMENT AND DEVELOPMENT METHODS

The embedded and firmware environment places extra difficulties on software development. This chapter tries to capture the unique characteristics in embedded programming-in-small.

A. Definition of Firmware

There is no common understanding of what an embedded system is, even though a number of different definitions exist. However, it is common in books about the subject to describe systems like air traffic control systems or space shuttles. To differentiate from this image the term firmware is used in this paper to mean the lowest level of system programming. Free Online Dictionary on Computing[1] defines firmware as "*Software stored in read-only memory (ROM) or programmable ROM (PROM). Easier to change than hardware, but harder than software stored on disk. Firmware is often responsible for the behavior of a system when it is first switched on.*" Programming at this level means that the developer must fully understand several constraints compared to their mainstream colleagues.

[1] http://burks.brighton.ac.uk/burks/foldoc/

B. Unique environment

Table I

Unique characteristics of firmware development.

Firmware Characteristic	Description
Culture of hacking	Still mostly "self-learned" electrical engineers as software developers. This often results in "*just software*" –syndrome.
Small teams and multiple hats	Single-person projects are common. The same person may be responsible for software and hardware design.
Co-design (distributed teams)	Strong dependency between software, hardware and also mechanics design.
One-time designs	Unique designs developing something new that does not exist before. Common to be build on top of latest technology.
Resource constraints	Limiting the recurring cost means that targets are usually the low-end microcontrollers.
Correctness and robustness	Firmware system needs execute its task correctly even in unexpected situations without interruptions. Systems may run for years without reset.
Lack of tools	For some targets the developers has to survive with just the simplest compiler and uploader.
Unconventional customer	Firmware may be developed for own hardware team as well.

Culture of hacking. Most people involved with firmware programming at the moment come from an educational background other than computer science (Pierce, 2004). They are mostly electronics engineers that at one point needed to write some code for a small microcontroller and ended up programming a little bigger microcontroller. Finally they became programmers, instead of electronics engineers. They didn't, and in most cases still don't have, a sound understanding of very basic software methods that are taught in elementary programming courses. Actually it has just very recently been understood that they even needed any of these. Many of the companies used analog electronics and then replaced some circuits with a microcontroller. Software is seen as coming automatically with the microcontroller. The cost of developing software is often overlooked.

Affected by the above, most organizations developing firmware suffer from *just software –syndrome*. Firmware code is not understood as real software, when in fact it is the extreme end of software development. It has to not only consider all the aspects that mainstream programmers do, but also deal with real-time requirements and struggle with them in an environment of very limited hardware resources. Senior management is still overlooking the software, which is today actually taking the same effort as electronics engineering. This is soon to change in favor of software. The old time hackers have been performing so well that it is hard for managers to understand that the software practices in the company do not match the new challenges of the firmware future. Kenn Orr compares such a manager to an alcoholic,

who furiously denies having any problems until the reality explodes in his/her face (Orr, 2002). Orr continues that in these situations it is common for a younger champion engineer to take over and correct the course of methods development.

Small teams and multiple hats. Often the software is developed by just one person, and it is possible that the same person is working with the hardware design as well. This results in limited communication, and instead heroism is frequently seen among these projects. Experiences and problem solutions are not shared, but instead knowledge is gathered slowly and individually.

Co-design. Designing a firmware system includes both hardware and software design (Wolf, 1994). Often also mechanics design is needed. In many cases the final hardware can not be defined until very late in a project. This means that most of the time the software development is done on different hardware than the actual target.
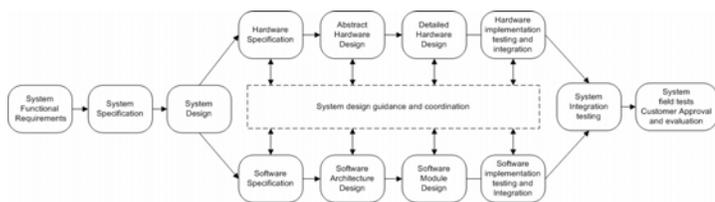

Fig. 1 Embedded system development in principle (Haikala and Märijärvi, 2000)

One-time designs. Firmware system projects are often very unique, one-time, designs. The project is creating something new, something that does not exist before. At the beginning of a project no one knows exactly what is going to be built, and thus no one has done it before. So under these conditions how would it be possible to create an iron bound requirements and specification document? The answer is that it is not possible. As Brooks already wrote in the 80's; "For the truth is, the client does not know what he wants. The client usually does not know what questions must be answered, and he has almost never thought of the problem up the detail necessary for specification. So in planning any software-design activity, it is necessary to allow for an extensive iteration between the client and the designer as part of the system definition" (Brooks, 1987). All this is especially true for firmware systems.

Resource constrains. Target microcontrollers usually have just a few kB of program memory and maybe 32 bytes of RAM. This forces the developer to understand constrains of resources. Often very hard real-time requirements are present. This means that timing has to be calculated on machine cycle level. If writing anything higher than assembler, the developer must also learn the compiler constrains.

Correctness and robustness. A system is correct when it does the right thing all the time. Such a system is robust when it does the right thing under novel (unplanned) circumstances, even in the presence of unplanned failure of portions of the system (Douglass, 1999). This means also fault tolerance. Usually a system needs at least to be able to get back on its feet in an intelligent manner. The need for hard resetting the unit should never occur.

Lack of tools. Even though the situation is much better today than it was few years ago, it is normal that in a firmware project the only tool is the assembler compiler and some way of getting the running code to target (Grenning, 2004).

Unconventional customer. The customer of a firmware project is not so clear case. There are at least three types of customer candidates for a firmware project. First there are sponsors, who usually have scope-setting and content-setting power on the project. They set the boundaries because they pay for the development. Marketing department usually plays this role. End-customers are the people that actually make use of your system. In a firmware project also a hardware team may be the customer for the software project.

One way to define the customer for the project is to ask few simple questions:

- Who defines the requirements for the firmware project?
- To whom it is delivered?
- Who accepts it and confirms that it meets its needs.
- Who pays for the development?

III. EMBEDDED SOFTWARE METHODS

Event though firmware development in many cases lacks any formal process, there exist also methods that are used. This chapter shortly presents four of them. They are chosen because they are mostly referenced in embedded software literature today.

A. Burn-and-pray or Build-and-fix

A recent survey of Embedded Systems Programming magazine[2] revealed that over 50% of firmware projects are still developed using a process which was described as "burn-and-pray". In literature (Douglas, 1999; Boehm, 1988) this is also called build-and-fix.

This model consists of only two phases:

1. Write some code
2. Fix the code

The obvious problem with this approach is that it produces

---

[2] http://www.embedded.com/

poorly modularized spaghetti code. Maintaining this code is a nightmare.

B. Waterfall, sequential

To help the problems with build-and-fix model the waterfall model was first put forward by Royce (Royce, 1970) and until early 1980s, it was the only widely accepted life-cycle model. A simplified presentation from Royce's model is presented in Fig. 2. It takes the process and divides it into separate phases which follow each other. Phases often include, but are not limited to, requirements, analysis, design, implementation and integration. It is a highly document driven model. Proceeding to the next phase requires a certain exit criteria (documents in practice) to be fulfilled. The problem with a sequential model is that it inherits to some degree from other fields of engineering and production process models. This means that it doesn't actually take software's unique nature under consideration. Brooks states that the inherent properties of software are complexity, conformity, changeability, and invisibility (Brooks, 1987). In that sense traditional engineering principles do not quite meet the unique needs of software.

C. Incremental

Since waterfall model, several variations have evolved to correct the limitations like rapid prototyping, incremental and spiral life-cycle models. The idea of incremental development is also presented in Fig. 2. Several full process rounds follow each other producing a product with more functionality. It is surprisingly often forgotten to mention that Royce already presented an incremental idea in his paper, when he stated that 'do-it-twice'. By that he meant that in parallel to larger project design and analysis phases a mini-project should be executed to speed up the learning process. Artifacts from this mini-project should be used to support the actual project phases.
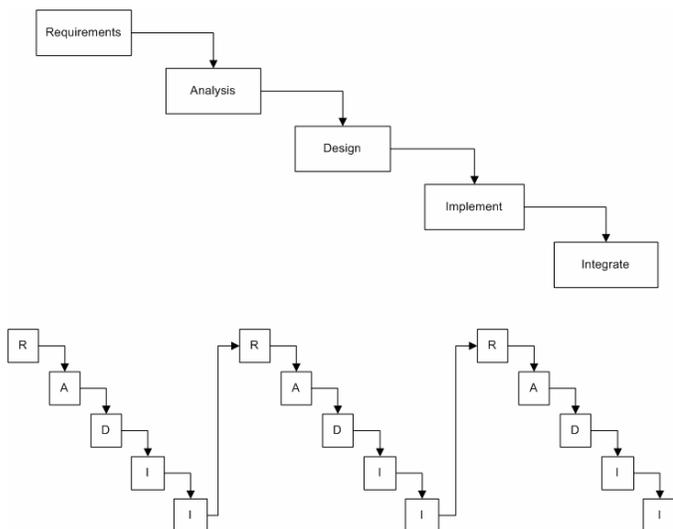

Fig. 2 Traditional models, sequential model above and incremental model below.

D. ROPES

Barry Boehm presented a spiral software model. In his paper (Boehm, 1988) Boehm suggests the model's suitability to software-hardware system development. Bruce Douglas has taken this challenge and developed Rapid Object Oriented Process for Embedded Systems (ROPES) (Douglas, 1999). The process model is illustrated in Fig. 3.
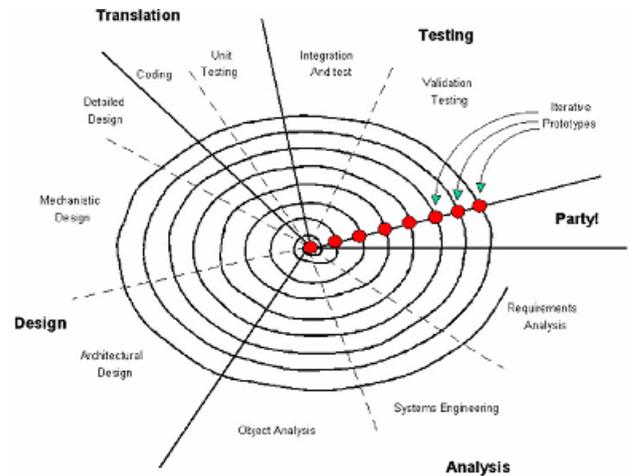

Fig. 3 ROPES development model

The ROPES process can be conceptualized as occurring in three different scales or time frames; the macro-, micro- and nanocycle. The macrocycle occurs over the course of many months to years and guides the overall development. It has four primary, but overlapping, phases; focus on key concept, focus on refinement of concepts, focus on design and implementation, and focus on optimization and deployment. The microcycle is completed in four to six weeks. The result of each microcycle is the production of an iterative prototype. The nanocycle is the most limited and usually lasts from 30 minutes to a single working day.

The ROPES spiral is different from most other spiral processes in a couple of ways. First, the system engineering subphase is included in the analysis phase. This phase is addressed to help in projects where software-hardware co-design is needed. The party phase is where initial project planning takes place, as well as the ongoing process improvement and project redirection. The requirements phase usually tries to hammer the overall project requirements at first spiral, but requirements also evolve throughout the project. Requirements analysis is done by using use cases. Only a few use cases however are explored in greater depth in each spiral.

The model puts strong emphasis to UML modeling. Douglas presents a long list of artifacts to be produced in each

phase of a spiral. In his work Douglas uses examples of huge systems, like air traffic control systems. He describes in detail methods to distribute threads and describes the usage of design patterns[3] in design phase. By doing this the process description highly concentrates on analysis and design methods.

The ROPES process is said to be highly scalable by its author, and claimed to be suitable for projects ranging from a single person to several hundred developer projects. Scaling is done by limiting the produced artifacts, but this is not described in detail.

E. RUP SE

Rational Unified Process (RUP) is a process framework developed by Rational Software Corporation. It divides the development process into four phases; Inception, Elaboration, Construction and Transition. Each phase is executed in several iterations as shown in Fig. 4.
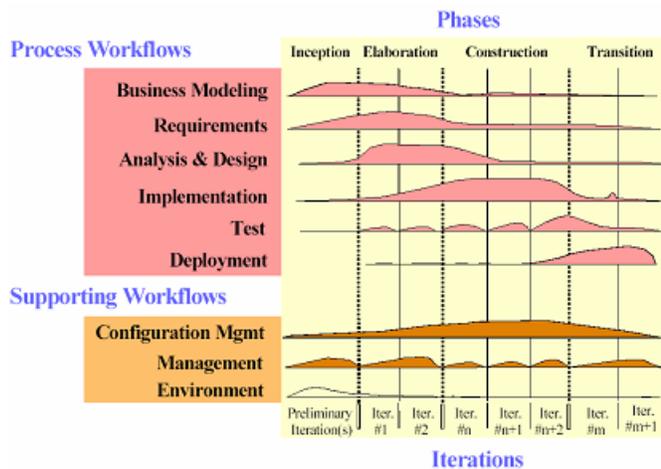

Fig. 4 RUP development model

Inception.   High-level view of the project is created. Abstract system architecture is also created and the scope and boundaries of the project are established.

Elaboration. The whole architecture is implemented for the first time. At the end of the phase the feasibility analysis is done to project vision, architecture and project plans.

Construction. Most of the implementation is done in incremental fashion. One or more releases are done to get feedback of the course of the project.

Transition. Software is tested and released to customer. First release may have beta status. More releases may be done to correct the problems found. Customer documentation is also produced during transition.

RUP framework describes the key process elements,;roles, activities, artifacts, workflows, disciplines and relationships among these. The base RUP framework contains tons of information and it is supposed to be customized to use by its customers. Framework is supported by extensive CASE toolbox. Customization and configuration can be done by using RUP plug-ins. A plug-in can bring new elements to RUP and/or it can redefine the existing elements. One of the plug-ins for RUP framework is SE (Systems Engineering) plug-in, which was made available in 2002.

RUP SE addresses projects that (Cantor, 2003):
- Are large enough to require multiple teams with concurrent development
- Have concurrent hardware and software development.
- Have architecturally significant deployment issues OR
- Include a redesign of the underlying information technology infrastructure to support evolving business processes.

RUP SE supplements RUP with additional artifacts, along with activities and roles to support the creation of those artifacts.

F. Other methods

Software method is different from process model as it only describes how to execute a certain phase of development. Some of the methods used in software development have also already existing real-time extensions. These include structural analysis/structural design (SA/SD) and Unified Modeling language (UML). Both of them have real-time extensions as SA/SD-RT (Ward and Mellor, 1986) and UML-RT used in Rational Rose design software by Rational Software Corporation.

G. Summary of Embedded Methods

All presented methods have limitations in firmware development. First two perform poorly in any complex software project with changing requirements. The latter two address the system development specifically. However these are developed larger teams in mind. This is seldom the case in firmware development. Table II summarizes the methods.

---

[3] Design Patterns are recurring solutions to software design problems.

Table II.

Summary of embedded methods.

Method	Amount of documentation and experience reports	Scope	Specific to method	Limitations
Build-and-Fix	Large	Not defined	No rules.	Poor results in any larger project.
Waterfall	Large	Software project management.	Strict rules on documentation. Proceeding in process requires certain artifacts to be developed.	Problems in turbulent environments with rapidly changing requirements.
ROPES	Small	From enterprise framework to engineering practices.	Party phase added to spiral model. Process is evaluated during this phase.	Scaling to smaller projects is not described in detail.
RUP SE	Large	Framework for software development and project management	CASE tool support. Emphasis on modeling.	Targeted to large software with multiple teams. Needs knowledge when configured to smaller environment.

IV. AGILE METHODS

The so called agile software models take the incremental process model even further, the increments are extremely short, and all life-cycle phases are advancing in parallel. Beck (Beck, 1999) describes this transition from more traditional models to agile as turning the process model sideways. Agile methods have gained a lot of attention in the software community during the past few years, but the movement is a decade old. Today the ideas have gained foothold also in the industry among the practitioners and not just among the academic crowd. Agile comes in many flavors. In this chapter the agile philosophy and the most documented Agile Methods, their process and practices, are reviewed.

A. Agile Manifesto

Several methodologists, all proponents for lightweight process models, as they were previously called, got together in Snowbird, Utah February 2001 to talk about what their methodologies have in common. As a result they agreed that better common name for their ideas than 'lightweight', would be 'agile'. They went on to form *Agile Alliance*[4] and they wrote down their ideas as the *Agile Manifesto*[5].

Agile manifesto states:

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.

Alistair Cockburn, who participated the Utah meeting and who is the author of the Crystal Family methodologies elaborates the manifesto in his book (Cockburn, 2002). The first sentence is telling us that these practices are not invented by the Agile Alliance, but merely "uncovered" while actually practicing the art of software development.

The first value states that the most important component in software development is the people doing it. "What the first value expresses is that we would rather use an undocumented process with good interactions than a documented process with hostile interactions."

The second value gets the focus to working software. After all you can't ship the design, if the customer has requested software. The working system is the only thing that tells you what the team *has* built. Documents can be very useful, but they should be used along with the words "just enough" and "barely sufficient". If you are not generating the code from

---

[4] http://www.agilealliance.com
[5] http://www.agilemanifesto.org

your models they are used for communication and they do not need so much detail.

The third value describes the importance of collaboration, the dialogue between the customer who wants the software and those who are building the software. Although contracts are useful at times, collaboration strengthens the development when a contract is in place and when no contract exists.

The final value might be seen as the key point of agile methods, since it states the importance of being ready to react to change. Change is almost inevitable in software development projects. Change may happen in business, staff, schedules, priority and so on. Agile methods use short increments to develop code. It is common that between these increments the management change the course of the project, as can be seen in Fig. 5.
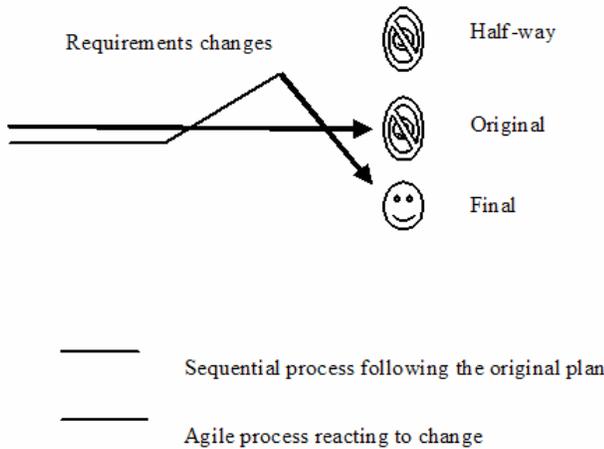

Fig. 5 Describing the effect of "welcoming the change" (Cockburn, 2002)

The closing sentence points out that Agile Alliance is not saying that software has been developed badly in the past. Agile Alliance still recognizes the value of tools, processes, documentation, contract and plans.

Agile alliance further agreed on 12 principles, listed in Table III.

Table III.
Agile manifesto practices

Agile Manifesto Principle	Explanation
Satisfy customer through early and frequent delivery of valuable software	You can't ship specifications or UML models if the customer has ordered software.
Welcome changing requirements even late in the project	Rather than resist change, the agile approach strives to accommodate it as easily and efficiently as possible.
Keep delivery cycles short(e.g., every couple of weeks)	Each delivery should have some additional value to customer, but *deliver* is not the same as *release*.
Business people and developers work together daily throughout the project	High level view of requirements is not enough for design and coding, so the gap is closed with frequent interaction between the business people and the developers. People who are authorized to make decisions are usually also more motivated.
Build projects around motivated individuals	People are the ultimate success factor for a project. Decisions must be made by the people who know the most about the situation.
Place emphasis on face-to-face communication	In addition to the lack of documentation, the *lack of understanding* is common cause for problems. Even though programmers write code, they are not writers.
Working software is the primary measure of progress	Iterative development provides milestones and accurate measures of progress. By delivering often the details of requirements can be captured.
Promote sustainable development pace	Hacking puts glamour to long nights and weekends, but agile methods need alert people and those long nights don't actually provide greater productivity anyway.
Continuous attention to technical excellence and good design	Good design is fundamental since the design is continuous activity in agile project.
Simplicity – the art of maximising the amount of work not done - is essential	In an agile project, it's particularly important to use simple approaches, because they're easier to change. It is easier to add something to something simple, than to take away something from something that is complex.
The best results emerge from self-organizing teams	The best architecture, requirements and design emerge from teams in which interactions are high and the process rules are few.
Team reflects regularly where and how to improve	An agile team continuously refines its process and methods to improve and to match the changing circumstances.

B. XP (Extreme Programming)

Kent Beck, the chief evangelist for Extreme Programming (XP), describes the methodology as "lightweight methodology for small-to-medium-sized teams developing software in the face of vague and rapidly changing requirements." (Beck, 2000; Beck and Fowler, 2000) XP puts together the *12 known best practices* for software development, and takes them to extreme levels. These practices explore good old principles like code reviews, testing, architecture design etc. in a new way. "The practices support each other. The weakness of one is covered by the strengths of others", Beck goes on.

The philosophy and the 12 practices are wrapped around *four values; Communication, Simplicity, Feedback and Courage*.

Communication. Beck says that problems with projects can invariably be tracked to be caused by lack of communication. Many of the XP's practices, *metaphor, planning game, pair-programming and coding standard,*

promote communication between programmers, customers and managers.

Simplicity. The second value can be summed in question "What is the simplest thing that could possibly work?" XP team will do just that with their *simple design* practice. According to XP philosophy it is better to do a simple working thing today and pay a little extra in the future to change it, than it is to make a complicated, reusable thing and pay for it today, but maybe never use it in the future.

Feedback. XP emphasizes the effect of feedback. Feedback is given all the time during the project. Programmers get feedback in minutes or even seconds from their *unit testing*. *On-site customer* gets feedback about new user stories immediately in form of effort estimates. The product is put to production in *small releases* as soon as possible to get feedback and so on. *Collective code ownership* and *continuous integration* furthermore help to get immediate feedback. Feedback of course is strongly related to good communication.

Courage. The team needs courage to execute all the XP practices in their extreme level. Programmers should have courage to *refactor* even when it means a heavy architectural change. They should have courage to throw away bad code, even if it works.

XP also promotes sustainable development speed. The *40-hour week* supports this.

By being part of an Extreme team, you sign up to follow the rules. But they're just rules. The team can change the rules at any time as long as they agree on how they will assess the effects of the change.

C. SCRUM

(Schwaber and Beedle, 2002) say that their process model, Scrum, expects every process to be unexpected. By that they mean that Scrum adapts to every situation and reacts to any change with natural feel. Scrum tackles the management problem of software development. As it does not address any particular software development practice it can work as a framework. Schwaber and Beedle express this as "if practices were candy the Scrum is the wrapping paper for candy".

Scrum divides the development into 30 day long increments, called *Sprints*. Each sprint is planned in a *Sprint Planning Meeting* and the results of a sprint are demonstrated and feedback is given in *Sprint Review Meeting*.

The role of requirements management is played by the *Product Backlog*. The Customer and chosen *Project Owner* will modify and prioritize the Product Backlog during the whole project.

Sprints are guided with *Sprint Backlogs* which are drawn from Product backlog for each sprint at the time. How the actual sprint is to be executed is left completely to *Scrum Team* to decide. The *Scrum Master* takes the role of a coach of the Scrum team. This is normally filled by a person who is traditionally called a project manager or a team leader.
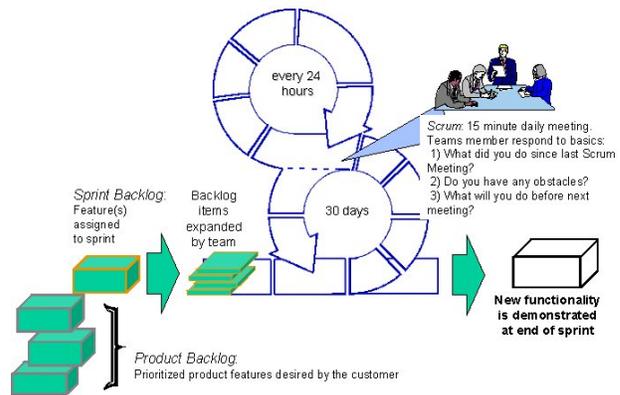

Fig. 6 Scrum[6].

D. The Crystal Methodologies

Alistair Cockburn highlights the fact that no one process size fits all projects (Cockburn, 2002). He models this with a diagram where the horizontal axis presents the number of people. These categories are shown with different colors. In his book Cockburn explains that the colors continue Clear, Yellow, Orange, Red, Magenta, Blue, Violet, and so on. This is due to the metaphor of crystals. Moving right on the horizontal axis, towards a darker color, means more people to coordinate and thus a need for a heavier process. Moving up on the vertical axis means more severe possible loses caused by a failure in the system under development. This means use of more rigor process and more ceremony, meaning more formal practices. The levels of possible loses are **C**omfort, **D**iscretionary Money, **E**ssential Money and **L**ife.
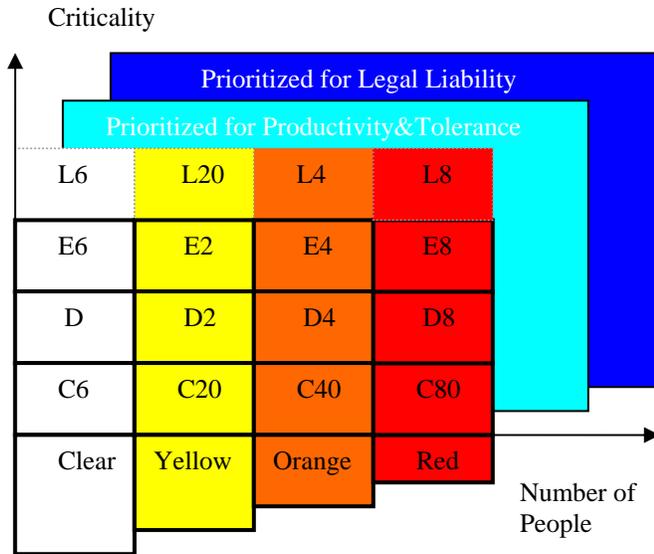
[6] http://www.controlchaos.com/

Criticality

Fig. 7. The crystal family of methodologies.

In his article (Cockburn, 2002b) Cockburn widens the diagram to consist team sizes up to 1000 persons. He also introduces a third dimension to the diagram as planes which present the idea that projects run to different priorities, some prioritizing for productivity, some for legal liability, and others for cost, predictability, agility, and so on.

The two rules, core elements, common to the Crystal Family are; the project must be incremental development and the team must hold pre- and post-increment reflection workshops.

The sample methodologies given are tailored by the team while working on a project. For this purpose the two base techniques in Crystal are the *methodology-tuning technique* and technique for holding *reflection workshop*.

Substitution of elements from similar methodologies is permitted. For example, the team could decide to use Scrum timeboxing and dynamic prioritization policies, daily stand-up meetings, pair programming from XP, and so on (Cockburn, 2002).

E. Adaptive Software Development

Highsmith presents guidelines for managing large, complex projects with adaptive approach (Highsmith, 2000). He presents lots of things that could help the project to succeed, but does not address the "how" part of the problem in depth.

The process is guided with three phases; *Speculate*, *Collaborate* and *Learn*. He also emphasises the fact that larger, more complex, projects need more rigor. Further he embeds the adaptive approach to common phase-gate process model by replacing workproduct with term *workstate*, to describe partial development of an item.

Highsmith gives five characteristic of ASD.

- Adaptive cycles are mission-driven
- Adaptive cycles are component-based
- Adaptive cycles are iterative
- Adaptive cycles are time-boxed
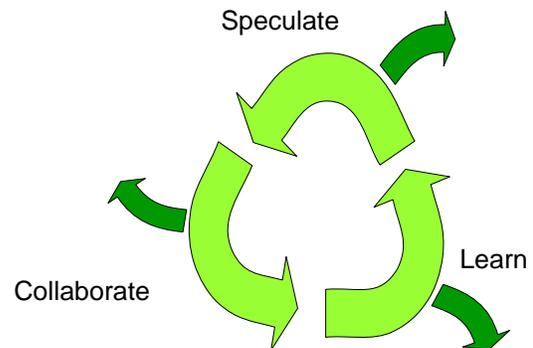- Adaptive cycles are risk-driven and change-tolerant

Fig. 8 ASD

Speculate. Planning becomes speculating, and it is most likely to be wrong anyway. Deviations guide us toward the right solution. The word "speculate" indicates that we are planning to the best of our ability at the moment.

Collaborate. In contrast to collaboration, communication is passive – it transfers information, with an intent to inform. Collaboration is active – it requires active participation, with an intent to add value.

Learn. ASD promotes learning trough different ways of feedback loops. Learning means gaining mastery through experience.

Workstate. ASD differentiates from phase-gate style process models in that instead of requiring a certain work product to be ready at a milestone or gate, an additional parameter is given to all work products. This is the state of the product. This way the model can use partial information to proceed. Highsmith proposes five states to be used; Outline(Conceptual), Detail(Model), Reviewed(Revised) and Approved(Available).

F. DSDM

Dynamic System Development Method (DSDM) is a

framework for rapid application development (RAD) and is maintained by the DSDM consortium[7]. DSDM process consists of five phases:

- Feasibility study
- Business study
- Functional model iteration
- Design and build iteration and
- Implementation

The last three phases are iterative and incremental. The fundamental idea behind the DSDM is to first fix the amount of resources and time, and then scale the functional requirements to that.

DSDM puts weight to prototyping. The first prototype may be created as early as in feasibility study phase. Business study phase will produce a prototyping strategy and during the next two phases the prototype will evolve and be constantly reviewed by the customer. The development takes advantage of timeboxing techniques.

Development roles in DSDM contain developers, team leaders, technical coordinator and project manager. Team leader is basically a more experienced developer while the technical coordinator is responsible for the software architecture and quality issues. Project manager can come from user community or from IT.

DSDM is said by its authors to be more suitable to business domain than to engineering or scientific applications.

G. *Feature-Driven Development*

Feature-Driven Development (FDD) (Palmer and Felsing, 2002) is constructed of five sequential processes:

- Develop an overall model
- Build a features list
- Plan by feature
- Design by feature and
- Build by feature

Agile development is supported in the last two processes, design and build by feature, where iterative development of a system is done. The key roles in FDD include project manager, chief architect, development manager, chief programmer, class owner and domain experts. A person can have multiple roles in a project, or a single role can be played by number of people. It is worth to mention that the chief architect does not just deliver the architecture to programming team. Instead he stays with the project until the end, and is responsible for all final design issues.

FDD differs from other agile methods in that according to (Palmer and Felsing, 2002) it is also suitable for critical systems.

[7] http://www.dsdm.org/

H. *Cycles of Control Framework*

The Cycles of Control Framework is intended to combine business and process management through Four Cycles of Control (Rautiainen, Lassenius and Sulonen, 2002). The cycles and related time scales are illustrated in Fig 9.
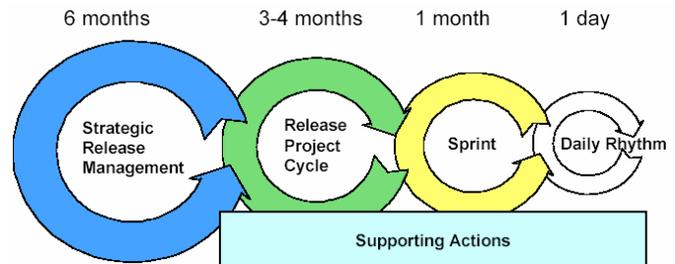

Fig 9 The cycles of control framework (Rautiainen, Lassenius and Sulonen, 2002))

The size of cycles represents the timebox which one of these four cycles is taking. The bigger the radius is the longer is the control period.

Strategic Cycle. The leftmost cycle is acting as an interface between business management and software (or system) development. It contains the organization level long term planning for scope and timing of product releases. Strategic Cycle is where major technology decisions are made. The product line of an organization is defined here, as is the overall vision statement for a single project. Practices like XP's Metaphor or Scrums Product Backlog may be used in a Strategic Cycle. Strategic cycle creates a release project plan or a product and technology road map.

Release Cycle. The next control cycle focuses on a single release. The main activity is to create a more detailed plan to execute a release defined in the previous control cycle. This includes making a time schedule for iteration cycles needed to get the prioritized features in to the product. The Cycles of Control framework allows flexibility for choosing either to design for features or design for deadline. The Release Cycle is the phase where this decision is made. Release management may prolong iterations or add people to the project as seen proper. Release cycle creates plans for iteration schedule and methods of reporting to Strategic Cycle.

Iteration Cycle (Sprint). The function of an Iteration Cycle is to produce a working product with added value to previous version. Within iteration cycle the requirements from

the previous cycle are broken further down to small tasks to be completed by developers. At the end of each iteration there should be a stabile, working product. Demonstrating this to former cycles will provide better means to get feedback of the work done. It also offers an opportunity to take course correction if there have been changes in market or changes in priority of features.

The self-organizing team is enforced by the framework. The team is given the freedom to choose the working methods inside the iterations. This is a familiar approach by most of the agile methods, emphasized for example by XP and Scrum. Requirement freeze is a very basic practice with 30-day sprints in Scrum.

Time for this cycle is one to three months. In a rapidly changing environment iterations longer than this may result in project getting out of control, or too much away from the course. The short iteration cycle makes it possible to freeze the requirement set, at least for the developers. This will allow them to focus on the task at hand, and gives more trust to work without the fear of back-and-forth changing requirements.

Heartbeat Cycle (Daily Rhythm). To allow even more precise control over the process, the iteration is divided into smaller tasks, mini-milestones. These are the Heartbeat Cycles. They should typically be limited to few working days. The basic idea can be seen in daily builds as in Microsoft's synchronize-and stabilize or XP, or as daily stand-up meetings like in Scrum.

I. Other Agile methods

Pragmatic Programming. Pragmatic Programming (Hunt and Thomas, 2000) is a collection of best practices for everyday programming activity. It contains all together 70 tips for software engineering practices. The tips follow the principles of Agile methods, like incremental development.

Agile modeling[8]. Although not a complete process model, agile modeling is worth a short introduction since it can be used as a supplement to other agile methodologies. Agile modeling applies the barely good enough –approach to software modeling. It is built around five values; communication, simplicity, feedback, courage and humility. First four follow the XP values. Humility is included to point out that all the stakeholders in a project are different, but should be treated equally important to the project.

J. Summary of Agile Methods

Changing to agile methods means a change in way of thinking. While sequential models view change as a negative occurrence, the agile teams sees change as an opportunity to quickly answer to competitors change or to make a change in its own course which the competitor has to answer. Agile team

welcomes the change. Agile methodologies however differ to some degree. Abrahamson et al. used 5 lenses to differentiate the methods; software development life-cycle, project management, abstract principles vs. concrete guidance, universally predefined vs. situation appropriate and empirical evidence (Abrahamson et al., 2003).

---

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

Table IV.

Summary of agile methods.

Method	Amount of documentation and experience reports	Scope	Specific to method	Limitations
XP	Large	Programming practices	Pair-programming	Narrow scope. Extreme approach.
Scrum	Large	Project management framework	Daily scrums	No engineering practices.
Crystal-Family	Small	Scales according the project	Collection of base models based on project size. Methodology tuning techniques.	Not fully documented yet.
ASD	Small	"Mental" framework for large project management.	Work state instead of work product.	Describing mostly the 'what', not the 'how'.
4CC	Medium	Enterprise level framework	Interface between development and business models.	No practices, needs supporting methods.
DSDM	Medium	Project management framework.	Specifically talking about prototypes. Technical coordinator	Membership required to access full resources.
Feature-Driven Development (FDD)	Small	Design and implementation practices	Chief programmers Chief Architect	Limited practices, needs supporting methods.

It can be seen that there is a huge versatility of different software methods that are included under the term agile. Most methods also encourage to modify the base method for own use. The methodology adapted to an organization may be a combination of several so called agile methods, e.g. a scrum model for management issues supplemented by methods like Agile Modeling or Pragmatic Programming for development phases of software life-cycle. Extreme programming practices may also be included inside such a model, even if XP is not fully adapted. To map the software development to organization level and business model, the idea of Cycles of Control Framework can be used to work as an interface between these two perspectives. This idea is presented in Fig. 10.
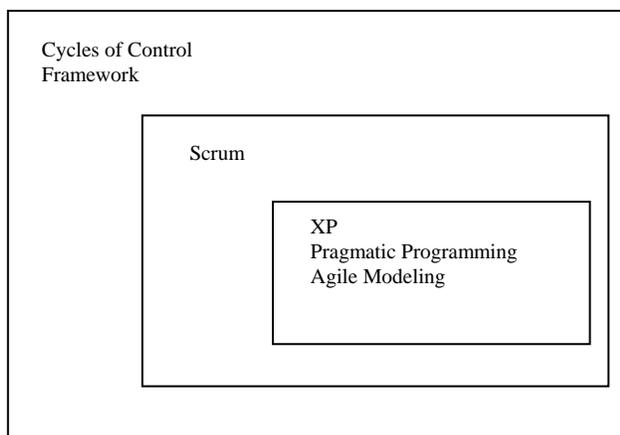

Fig. 10 The idea of wrapping agile methods.

V. AGILE METHODS AND FIRMWARE DEVELOPMENT

Firmware development places some unique challenges to software development, but most of the problems are familiar from other areas of software as well. Common problems include frequently changing requirements, fast time-to-market and high quality requirements. These problems are tackled with agile methods. In this chapter the compatibility of agile methods and firmware development is analyzed.

A. Existing Literature

Dan Pierce sees opportunities in adapting XP into embedded system programming (Pierce, 2004). He examines the adaptation of the XP methods by evaluating four XP activities; *planning*, *designing*, *coding* and *testing*.

In *planning* activity he states two possible needs for adaptation. First of all the co-design characteristic of embedded systems may restrict iterative process. It may not be feasible to do iterative design on hardware because it is much more difficult to change at later stage. Second the prioritized iterations in embedded system design usually means writing the hardware drivers first. There exist good reasons for that:

1. The real-time constrains are usually the most uncertain to be fulfilled, and most often the hardware drivers are associated with these.
2. There may be hardware that is used for the first time. Again this is the most uncertain part and must be evaluated first.
3. Testing on real hardware means that there must exist some ways of giving inputs to the system and getting indication from the system. This requires that the system is able to drive the hardware.

Designing in XP fashion concentrates on what is needed today and not what could be needed tomorrow. Many firmware programmers have just learned things like common interfaces and modular design. In firmware this means for example writing a hardware driver for a peripheral so that it could be used by others as well. This results in increased costs in form of development time, code size and code execution time. "This investment only pays off if the driver is used on

another project in a different way at a later time." Pierce suggests that firmware developers should adapt a style with obvious hooks on how to generalize later.

Pierce recognizes couple of pitfalls in firmware development and over designing. Firmware developers have learned to divide the lowest level of a driver (physical layer) from the non hardware-specific code. The first pitfall is that this may get out of hands and result in a too deep architecture. The second pitfall according to Pierce is too anxious need for use of a commercial RTOS. In simple systems this will have a negative effect and will make the design more complex, the opposite to XP's approach "What's the simplest thing could possibly work?"

Coding in a XP project should be started as soon as possible. Pierce says "(Iteration) …should be implemented immediately by everyone who writes software even if you adopt nothing else." As stated earlier a usual characteristic of a firmware project is that the target hardware must we waited for. There are few options for a firmware team to tackle this:

1. Write the code and just compile and run a static verifying on that
2. Use a simulator on a PC to run the code. Prices on these have lowered dramatically and for small microcontrollers you will get it for free.
3. Get an evaluation board. From this you can jump-wire to which ever electric circuits are available.
4. Assemble a breadboard version of hardware with essential electric circuits and possibly use an in-circuit real-time emulator. This may be even a better solution than an actual target PCB with limited layout space for effective hardware debugging.

Probably the most famous practice of XP is pair-programming. As stated in characteristics of firmware project, they are often designed by a single person. You can't form a pair, if you only have single person. Pierce suggests that pair-programming is still scheduled regularly with another developer from another project. This can be seen as a certain form of software review, just like pair-programming in the first place.

XP says about *testing*, that if the code does not pass all the tests, the code does not exist. All the code should be verified through automated regressions tests. Again Pierce sees a couple of opportunities to adapt this to embedded development.

1. Using a serial port to report test results from target hardware. Even the low-level microcontrollers usually have a serial port peripheral on them and reporting via that has low overhead in code size.
2. Using a simulator for testing. This does not allow testing the co-operation of more complex hardware

that will be present on final design, but have capabilities for automated tests with I/O stimulus patterns saved on hard-disk.

James Grenning has also analysed (Grenning, 2002; Grenning, 2004) using XP practices in embedded software project. He addresses the benefit of getting an early start in software development using XP practices. At the beginning of the project software developers do exploration. By doing that they get the best guess on some set of basic features to be developed and they develop the first release plan. The goal is to quickly move to development.

Grenning uses development of small embedded systems, such as a telephone switch, as examples. Evolution of a design and architecture is done with UML modeling and implemented in Java, strongly relying on interface design.

As a conclusion Grenning says that embedded systems development can benefit from the practices of XP, but he also sees problems. One major problem in using XP or TDD (Test-Driven Development) in embedded systems development is the lack of tools. "Deploying embedded software may still have high costs, but XP can help to lower the cost of change for much of the embedded software development cycle."

(Grenning, 2004) gives a warning that he may have simplified the problems of embedded systems development and reminds that there are additional problems to be addresses in more depth. These problems include issues of concurrency, timing constraints, testing a large application and how this fits in the bigger picture. These problems he promises to address in future papers.

In their experience report from telecom projects using agile methods, (Vanhanen, Jartti and Kähkönen, 2003) came up with some interesting ***success factors, new practices and other findings,*** which are likely to apply to firmware development as well. The research group studied three software projects for their agile methods by interviewing the team members and project managers. The three projects varied from 50 man years to 6 man years and team sizes from 4 to 12, and included a project for integrating and porting of embedded software.

Success factors included software architect's continuous involvement during the life-cycle, competent developers and managerially and technically competent project manager, and frequent team meetings.

New practices included a *Technical Authority*, which means a technically competent project manager. In one project under research a team member became the project manager during the project. This worked as a success factor for this particular project. "The project manager's understanding of the technical details increases the likelihood to have the courage to decrease control elements such as managerial documents and project reviews from the process." (Vanhanen, Jartti and Kähkönen, 2003). Another new practice found was *Team Continuity,* e.g.

the importance of the key personnel staying in the project the whole time.

Other findings uncovered that firstly collective code ownership improves agility by removing delays in development. Secondly while the landscaped offices encourage communication they may disturb work that requires high concentration. Third it was found out that developers accepted an agile process well.

Rising and Janoff (Rising and Janoff, 2000) report three success stories with Scrum and small software teams. They did some modifications to the method, like not having the daily scrums, but instead had two to three short meetings a week. All three cases succeeded, and they found out that all three teams got to work together as a team and all three teams planned to continue using Scrum.

Rautiainen, Vuornos and Lassenius present results on a case study applying 4CC at Smartum, a small software product company (Rautiainen, Vuornos and and Lassenius, 2003). 4CC was used to structure the process development and the chosen process combined practices from Rational Unified Process (RUP), eXtreme Programming (XP), Scrum and Microsoft's synchronize-and-stabilize.

Process improvement did not initially succeed since management commitment was lacking and the improvement was initiated "bottom-up" by the developers. This resulted in that sprint logs could not be kept frozen during the sprints, and schedules slipped. After getting the management involved, the case study was evaluated to be an overall success. Effort estimations and collaboration to customers imporoved. Making the progress of development transparent has enabled management to make more informed decisions and confidence to schedules increased. "The case study shows that practices from different agile process models can be combined to make a coherent whole." The paper lists success factors to process development. First of all 4CC was seen to help as a framework to structure the process development. Secondly after the management commitment was achieved and developers themselves were still in key roles, improvement was started. Also among the listed success factors were understanding that icremental improvement takes time and the need for a driving force or champion (in Smartum the R&D team leader).

The working hours consumed in process development at Smartum in year 2002 were 120 hours by team leader and 49 hours by developers together. Data from management was not available.

Experience report on applying 4CC AvainTec (Vanhanen, Itkonen and Sulonen, 2003) describes how 4CC is supplemented with practices from other agile methods in another case. They applied Scrum techniques for project management (only 2 scrum meetings a week instead daily scrums) and some of the XP's practices for development. XP practices included automated unit tests, coding standards and modified pair programming. They chose to practice pair programming for any harder tasks, and otherwise close co-operation.

At AvainTec they also implemented a "red flag"-practice which means that any additional recovered tasks which does not affect to actual sprint goal can still be added to sprint back log as high priority tasks. A fixed 40% amount of effort is reserved for non-development work, which however included working on these "red-flags". The new process was welcomed well by both the business and the development.

B. Is Agility for Firmware?

Boehm and Turner (Boehm and Turner, 2003A) present criteria for project to evaluate whether it is on agile or plan-driven home ground. Table V presents these criteria and describes firmware ground.

Table V.

Agile and plan-driven home ground, and firmware ground. (Boehm and Turner, 2003A)

Project Characteristics	Agile home ground	Plan-driven home ground	Firmware ground
Application			
Primary goals	Rapid value, responding to change	Predictability, stability, high assurance	Projects are developing something new, rapid feedback from customer is essential
Size	Smaller teams and projects	Larger teams and projects	Single-person projects are common.
Environment	Turbulent, high change, project focused	Stable, low change, project and organization focused	Projects are developing something new; requirements are sure to change. Often applying new technology and evolution is based on new experience.
Management			
Customer relations	Dedicated onsite customers, focused on prioritised increments	As-needed customer interactions, focused on contract provisions	Unconventional software customers, e.g. hardware team
Planning and control	Internalised plans, qualitative control	Documented plans, quantitative control	Often looks nearly hacking without planning from outside. Planning comes from tacit knowledge.
Communications	Tacit interpersonnel knowledge	Explicit documented knowledge	Lots of hidden information gathered by experience
Technical			
Requirements	Prioritised informal stories and test cases, undergoing unforeseeable change	Formalised project, capability, interface, quality, foreseeable evolution requirements	Prioritisation comes naturally from OS and driver development. Unforeseeable changes because of uncertain requirements and new technology.
Development	Simple design, short increments, refactoring assumed inexpensive	Extensive design, longer increments, refactoring assumed expensive	Simple design is essential. Refactoring may be difficult because of real-time deadlines.
Test	Executable test cases define requirements, testing	Documented test plans and procedures	Final tests can only be run when hardware is available. Official tests are expensive.

Personnel

Customers	Dedicated colocated Crack[9] performers	Crack[5] performers, not always colocated	Variable customers from hi-tech hardware team to global marketing team.
Developers[10]	At least 30% full-time Cocburn Level 2 and 3 experts; no Level 1B or level –1 Personnel	50% Cocburn Level 3s early; 10% troughout; 30% level 1B's workable, no Level –1s	Often no CS-educational background. Limited software method toolbox.
Culture	Comfort and empowerment via many degrees of freedom (thriving on chaos)	Comfort and empowerment via framework or policies and procedures (thriving on order)	Close to pure hacking (chaos). Fire-fighting against changing requirements.

Primary goals for firmware project should be rapid value, and feedback from customer. Firmware projects are often developing something unique, and no one has a clear picture of what it is supposed to be like. This means that projects are extremely difficult to predict. The best way to get the evolution of the idea on its way is to build a prototype based on the best guess on requirements. Customers need to learn to understand and use this powerful tool. Same applies to developers to learn not to over develop prototypes before the first release, but instead deliver it for review and get feedback sooner.

The *size* of a firmware software project team is usually only few people and often they are just single-person projects.

Environment of a firmware project is usually highly turbulent. The nature of unique design, usually applying new technology, makes all the requirement decisions extremely unreliable. Every time the project staff learns more the whole set of requirements is at risk of changing. Learning happens for the whole development period.

Customer relations may differ in firmware project dramatically from mainstream software project. The position of customer may be filled by end users, product manager or hardware development team in some cases. It is obvious that the relationship between software developers and these different types of customers vary a lot. A global marketing department may possibly be located at the other side of the globe.

Planning and control in firmware project is usually extremely limited. The software method and technique toolbox of an average firmware developer is quite light. This is because he normally lacks formal computer science educational background. This results in poorly executed planning and control of a project. Nevertheless a seasoned firmware developer has a magic like ability to quickly develop an operating system framework with essential hardware

drivers as an abstract architecture of the system to be. This first shot is usually also very close to the final optimized solution. The seasoned developer has internalized plans for his software structure.

Communication seldom takes place if all the development is done in single-person projects, and instead heroism is normal. The knowledge is hidden from others in order to make oneself look better. Experienced developers have tons of tricks how to make their software more efficient on a certain microcontroller. Saving few bytes makes a difference between a successful and a failed project in this environment. It would be crucial to any project to share this existing experience based knowledge.

Requirements for firmware software include more non-functional requirements than software in general. These are the need to run code in a small microcontroller, hard real-time deadlines etc. Often these projects are also developing a new kind of device, and this means that at the beginning of a project the customer of the product has no way of knowing exactly how the product should work. He has only the best guess, which is usually very rough. Requirements evolve during the project when the team learns more about the technology being used and the device to be developed. Prioritization of requirements is usually quite straightforward. In order to get the system to do anything at all, an operating system and hardware drivers have to be developed. To see if the chosen technology is feasible, all the new technology dependant parts should be implemented in the beginning as well. If any hard real-time constrains are to be met, then these need to be analyzed as soon as possible.

Development of firmware is forced to very simple design. The resources of target microcontrollers limit the possibilities of using any higher level programming languages effectively. Execution time for certain parts of the code needs to be known on machine cycle level. This however also makes refactoring a risky business. If strict timing in a routine is optimized, then a small change to that part of the code may break the whole system.

Testing in a firmware project is trickier than in workstation software development. The resources, memory and timing form the highest risks to a project. These should be tested on real target, but this is not usually available until very late in the project. There may also be requirements to run official tests on firmware products. These need scheduling beforehand. They are also expensive, and should be run only once before the product is released.

Automated regression testing has also difficulties in firmware development. There is no program memory space to have testing code integrated. If compared to PC programming there is also lack of disk drives for storing the test cases and display units to show the results.

---

[9] Collaborative, representative, authorized, committed and knowledgeable.

[10] (Boehm and Turner, 2002) divide developers into 5 categories according their performance; -1, 1B, 1A, 2 and 3.

Customers vary from highly technically skilled hardware team to purely business oriented people.

Developers of firmware software are said to be skilled on a different aspect than say for example database programmers. In firmware project when a programmer saves a byte of executable code or cuts a microsecond from executing time he is considered skilled. In this fashion it is clear that firmware developers are not performing in skills that are normally learned in computer science studies.

Culture in firmware projects is at least very close to pure hacking. Projects are fired away in build-and-fix manner. The hammered MS Project Gannt chart is kept unmodified even while the work load increases everyday, because of multiple requirement sources. It is usual that project managers, not to mention senior management, lacks the knowledge of all the requirement changes sneaking in. In order to look good by their superiors, developers struggle to meet the original timetable despite the rapidly changing requirements. Firmware projects are clearly thriving on chaos.

In their article Boehm and Turner (Boehm and Turner, 2003B) also present a diagram for analyzing if a project is on agile or plan-driven home ground. Fig. 11 presents an imaginary firmware project using this diagram. The smaller the area of resulting shape is, the closer to agile home ground the project is. It can be seen that firmware is not clearly on agile home ground. Size and culture promote agile methods and requirement dynamism is also on agile home ground. The personnel are closer to plan-driven and criticality can be a strong discriminator against usage of agile methodology in a project. The criticality of firmware software can go to both extremes of the scale. It should be clear that modification to methodology is needed when moving from developing a DVD remote controller to developing a security system of a nuclear plant.
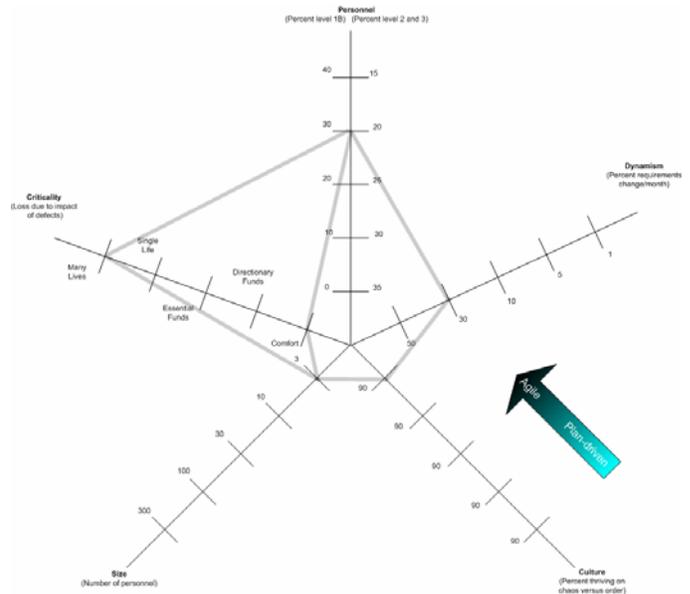

Fig. 11   Imaginary firmware project in home-ground diagram (Boehm and Turner, 2003B).

C. Adopting Agility

In Table VI the 12 principles from Agile Manifesto (see IV.A) are reviewed from firmware development perspective. Each principle is graded for its applicability with values Applicable, Needs Adaptation, Inapplicable. If the principle needs adaptation, the effort estimation is given on a scale of low, medium and high.

Table VI.			
Agile Manifesto Principles and Firmware development.			
Agile Manifesto Principle	**Applicability**	**Effort of Adaptation**	**Comments**
Satisfy customer through early and frequent delivery of valuable software	Needs Adaptation	High	Rapid value and immediate feedback is essential. However target hardware is not usually available.
Welcome changing requirements even late in the project	Needs Adaptation	Medium	Reality in firmware projects. Some changes however may affect hardware design or violate real-time constrains.
Keep delivery cycles short(e.g., every couple of weeks)	Needs Adaptation/ Inapplicable	High	Big part of firmware development is not visible to customer at all.
Business people and developers work together daily throughout the project	Needs Adaptation	Low	High cultural barrier between business people and developers custom to hacking.
Build projects around motivated individuals	Applicable		Agile methods should improve motivation among development oriented people.
Place emphasis on face-to-face communication	Needs Adaptation	Medium	Often distributed teams.
Working software is the primary measure of progress	Needs Adaptation	High	Firmware needs a technical person to evaluate the working status of software.
Promote sustainable development pace	Needs Adaptation	Low	Software ream may put glamour to long hours. Firmware development has some fixed dates (e.g. official tests, fairs).
Continuous attention to technical excellence and good design	Applicable		Thorough testing is essential for firmware with robustness requirements.
Simplicity is essential	Applicable		Simplicity is natural for real-time firmware because of limited resources.
The best results emerge from self-organizing teams	Applicable		Promotes team work. Firmware developers prone to heroism may lack these skills.
Team reflects regularly where and how to improve	Applicable		Important point for small organizations for adopting process improvement.

Satisfy customer through early and frequent delivery of valuable software. Incremental delivery is the best practice from agile methodology. This applies as well to firmware development as to any other type of software. There is lot of work to do to get the customers to understand the value and possibilities offered, but after that no one should be willing to go back to delivering all at once. In firmware development the target hardware may not be available at the beginning of the software project. In this case virtual prototypes running on a PC may be used. Evaluation boards may be used for more technical customers.

Some process models describe a very rigorous requirements phase. Firmware system project may however depend deeply on technology decisions. It does not make much sense making strong use case analysis if the chosen technology proves to be completely wrong two months later.

Prioritized development is what is usually done in a firmware programming anyway. You need to write some basic set of hardware drivers first to get the system to do anything visible. Real-time constrains has to be analyzed to be feasible at the beginning as well. The number of application features is usually quite limited. Anyway the customer can choose which drivers to write first. Which for example is more interesting; the indicator pattern or the delay caused by interconnection to other unit?

This basic set of hardware drivers and feasibility test for timing issues might very well fit to the first increment.

Welcome changing requirements even late in the project. Changing requirements are reality especially in firmware projects. They frequently apply new technology and develop

something new. Sometimes mistakes in hardware design are tried to fix in software, since fixing the hardware mistakes is more time consuming if a new circuit board is needed.

However welcoming these changes is not as trivial as in mainstream programming. Some changes may need modifications also to the hardware design. Refactoring may be risky in parts of code where the timing is done on machine cycle level. Adding executable code to that part may break the system completely.

Keep delivery cycles short. Short delivery cycles are an effective way of gathering feedback, which is essential for firmware projects. Applying this to firmware environment however needs innovative thinking. Lot of development work is done for example in error handling, EMC reduction and timing fine-tuning. These are things that are not visible to customers, if they are not highly technically oriented. The software team may also be highly dependant of hardware team for the next iteration. In these situations the software should be delivered as virtual prototype running on a PC, evaluation board or just prototype of actual hardware.

Business people and developers work together daily throughout the project. Since firmware development culture is still close to hacking, there is a high cultural barrier between the "hackers" and the "suits". A common language is mostly missing and attitude needs improving on both sides. Another possible difficulty is that the marketing department in the role of customer may be located in the other side of the world. One option to solve this is to shield the software team with technical project manager who will act as a customer proxy.

From agile view it is important just to have someone authorized to give immediate answers to any requirement questions the team might be facing.

Build projects around motivated individuals. People who focus on development get their motivation from different things than management oriented people (Baddoo and Hall, 2002). Working in a satisfying environment and getting appreciation from peer workers may be enough to motivate people. Agile methods promise to do this for team members.

Place emphasis on face-to-face communication. A firmware project always involves at least a software team and a hardware team. It is normal that these teams are distributed to separate floors of the building and in unfortunate cases the teams are located in different countries. In these situations video conference is the next best thing, if the people get used to using it effectively. Development tools, requirement management tools and software modeling tools are today also supporting collaborating. New communication tools are also tried, like e-mail, IRC groups and messengers. Results from these vary a lot from case to case.

Working software is the primary measure of progress. A firmware system may not do anything visible before the software is implemented as whole. There usually is no hardware to run the first iterations. This results in need of technical persons, and project manager/team leader, to evaluate progress.

Promote sustainable development pace. Firmware developers still put glamour to long hours, working on weekends and re-arranging vacations. Just like hackers tend to do. This however does not provide improved efficiency. Instead staying sharp in a job requiring innovative people should be the goal for management.

However scheduled, expensive official tests, or fairs, need deliveries on a fixed day with fixed functionality. These may put the team to work harder from time to time.

Continuous attention to technical excellence and good design. Thorough testing is a requirement for firmware systems, especially in critical environments. Even though many firmware projects need to be technically high level, people are not accustomed to place emphasis on design. Increasing size and complexity of firmware will however force to take this under consideration no matter which process model is followed.

Simplicity is essential. Simple design is natural for firmware development, since compilers and target environments don't support complex structures. However you may also witness two extremes in firmware projects. Some projects may be over designed, and work against the YAGNI[11] principle (Pierce, 2004). On the other hand global variables and single source file project can be seen as simple solutions in firmware crowd.

The best results emerge from self-organizing teams. Collaboration may be hard for old time hackers, who have tons of undocumented assembly tricks in their hat. These tricks have made them valuable for the organization, and the openness may be seen as a threat to this situation. These developers are extremely prone to heroism. Still self-organizing is quite natural for firmware development. It is the responsibility of the management to promote the team work and self-organizing. Appreciation by the peer workers is recognized as an effective motivator for development oriented people (Baddoo and Hall, 2002).

Team reflects regularly where and how to improve. This practice actually promotes bottom-up process improvement. This is a solution for small companies having difficulties affording resources to process and methodology development. Bottom-up approach also improves motivation for this type of activity in an organization (Baddoo and Hall, 2002).

D. Balancing the Methods

Highsmith (Highsmith, 2000) explains how the projects may fail even if they are on agile home-ground. If agile methods are blindly practiced for all activities in a project, the overall project may lose efficiency. Instead the stable predictable parts of a project may be driven with more rigorous methods, which usually are more optimized and thus more efficient. Also Boehm and Turner present methods to evaluate whether a certain environment and project is on agile or plan-driven home-ground (Boehm and Turner, 2003A). They suggest that different approaches are used on their home-grounds.

At the beginning of a firmware project it is needed to evaluate the required level of detail of up-front design.

E. Traditional vs. Agile FW Development

More than half of the firmware projects are executed with process model describes as build-and-fix. Process models have been developed to help the development of embedded systems in particularly. This paper reviewed the agile process models and reflected them to firmware development environment.

Table VII summarizes the introduced models in contrast to characteristics of firmware development.

---

[11] You Ain't Gonna Need It, principle of designing just what you need at the moment.

Table VII.

Comparing methods

FW characteristic	Build-and-fix	Waterfall	ROPES	RUP-ES	Agile Methods in general
Culture of hacking	✓ Chaos, hacking	✗ Order.	✗ Order	✗ Order, can be configured also lighter	✓ Thriving on chaos
Small teams and multiple hats	✓ Single-person projects	✗ Scales to larger teams	✗ Targeted to large, complex projects.	✗ Targeted to large teams.	✓ Targeted to small teams.
Co-design (distributed teams)	✗ No communication	✓ Communication based on delivered documents	✓ Addresses SW/HW composition	✓ Addresses HW and SW teams and co-design	✗ Requires co-located teams (FDD addresses teams)
One-time designs	✗ Difficulties when creating anything with significant size.	✗ All requirements supposed to be known at the beginning,	✓ Incremental development with prototyping	✓ Incremental development	✓ Incremental development, (DSDM with prototyping)
Resource constraints	✗ Poorly modularized spaghetti code	✗ Strong analysis and design phase before implementation. Difficult to analyze before trying.	✓ Strong analysis and design phase before implementation, Object-orientation causes size overhead.	✓ Strong analysis and design phase before implementation. Object-orientation causes size overhead.	✓ Emphasis on simple-design. Refactoring may be also seen as optimizing.
Correctness and robustness	✗ No support	✓ Independent analysis and design phase before implementation.	✓ Strong analysis and design. Can be scaled.	✓ Can be configured to more rigorous process.	✗ Problems with critical systems. (FDD address critical systems)
Lack of tools	✓ No need for special tools.	✓ No need for special tools.	✗ UML	✗ UML	✗ Requires automated tests, which are challenging to apply
Unconventional customer	✗ No customer collaboration	✓ Customer involved only at requirements phase.	✓ Strong requirements analysis phase is difficult to do with unknown requirements	✓ "Most of requirements" at elaboration phase	✗ Requires on-site customer. (FDD uses domain experts)

The table above is quite limited, as it takes the agile methods as a single process model. Some of the agile flavors address the problems individually. It can however be seen that there is no obvious solution for firmware development process model. Agile methods also have pros and cons.

VI. CONCLUSIONS

The embedded community has always been the last to adapt any novel software methods. The size and complexity of software has increased all the time, and this applies also to firmware development. This puts more demand on how the development work is executed. Currently most of the firmware development completely lacks a defined process. Process improvement in firmware development environment is at least different from PC software environment. The development has lots of extra constrains in addition to mainstream programming. However the crisis can still be avoided.

My first research question was whether so called agile methods can be applied to this different environment. The study showed that firmware development is not clearly on agile-home ground, but most of the practices could be adapted. Some need more adaptation effort and innovative thinking. Since all agile aspects do not fit the firmware environment the process model needs adaptation and balancing of different methods.

Several ideas to adapt agile methods to firmware development were presented. These included:

- Investment in up-front architecture
- Use of video conference and other new collaboration tools
- A customer proxy for developers
- Use of virtual PC prototypes

Many embedded systems are very *critical*, and may even cause loss of life in case of failure. It is only natural to want software for a pace-maker, or a space shuttle, from CMM level 5 -organization, and not from group of young propeller-heads using XP practices. This doesn't mean that there are not embedded applications which will suite much better the agile environment than the burden of heavy processes.

My second research question was whether the firmware development could benefit from agile methods. Experience

reports on agile methods are still scarce, and for embedded systems and agile methods it is nearly non-existent. There exist few papers about XP in embedded development. Firmware development has also common software problems, like fast time-to-market, changing requirements etc. These problems agile methods tackle as well in firmware development as in mainstream programming business. Experiences also show that small teams adapt to agile well.

Many of the methodologies under the agile umbrella can also be used for bottom-up incremental process improvement. This is an important point for small companies that can't afford to assign people to process improvement. It is easy to say that the firmware development can surely benefit from agile methods.

The author is looking for an opportunity to run a pilot project and analyze possibilities to agile adaptation based on experiences. Further research could study possibilities for adapting agile methods to embedded systems on system level. The first increment could include a breadboard version of circuits applying new technology, or just an executable simulation or simulation result document.

Software development is only a part of systems development. It differs from other activities due its unique nature. This creates an interesting research objective to map the activity in software development to this bigger picture. (Wallin, Ekdahl and Larsson, 2002) presented a model to map software life-cycle to business decision models. They present how to map stage-gate™-gate model used at ABB to RUP, Microsoft's synch-and-stabilize and extreme programming. Mapping is done by comparing software life-cycle milestones to gates of the stage-gate™ model. This perspective could be elaborated to systems development.

Even though not the silver bullet, agile methods are worth analyzing in firmware software development. *You should start trying openness by giving agility a fair chance.*

REFERENCES

(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.

(Baddoo and Hall, 2002) Baddoo N. and Hall T., Motivators of Software Process Improvement: An Analysis of Practitioner's Views, Journal of Systems and Software, vol.62, no.2, 2002, pp.85-96.

(Beck, 2000) Beck, Kent, Extreme Programming Explained, Addison-Wesley 2000.

(Beck, 1999) Beck, Kent, Embracing Change with Extreme Programming, IEEE Computer 1999, pp.70-77.

(Beck and Fowler, 2000) Beck, Kent and Fowler, Martin, Planning Extreme Programming, Addison-Wesley 2000.

(Boehm, 1988) Boehm, Barry. A Spiral Model of Software Development and Enhancement, IEEE Computer, Vol. 21, No.5, 1988, pp.61-72.

(Boehm and Turner, 2003A) Boehm, Barry and Turner, Richard, Balancing Agility and Discipline: A Guide for the Perplexed, Addison-Wesley, 2003.

(Boehm and Turner, 2003B) Boehm, **B**arry and Turner, Richard, Using Risk to balance Agile and Plan-Driven Methods. IEEE Computer Society, June 2003, pp. 57-66.

(Brooks, 1987) Brooks, F., No Silver Bullet: Essence and Accidents of Software Engineering, Computer, Apr. 1987, pp.10-19.

(Cantor, 2003) Cantor, Murray, Rational Unified Process for Systems Engineering. Part I: Introducing RUP SE Version 2.0. IBM Rational Software 2003. Available at http://www.therationaledge.com/

(Cockburn, 2001) Cockburn, Alistair, Agile Software Development. Addison-Wesley 2001.

(Cockburn, 2002) Cockburn, Alistair, "Learning From Agile Software Development – Part One", Crosstalk, Vol. 15, No.10, 2002, pp.10-14.

(Douglas, 1999) Douglas, Bruce Powell, Doing Hard time, Allison-Wesley 1999.

(Gibbs, 1994) Gibbs, W., Software's Chronic Crisis, Scientific American, Sept. 1994.

(Grenning, 2002) Grenning, James W., XP and Embedded Systems Development, 2002 Available at http://www.objectmentor.com/home

(Grenning, 2004) Grenning, James W., Progress Before Hardware, 2004 Available at http://www.objectmentor.com/home

(Haikala and Marijärvi, 2000) Haikala, Ilkka and Märijärvi, Jukka, Ohjelmistotuotanto. Satku, 7th edition, 2000.

(Highsmith, 2000) Highsmith III, J., Adaptive Software Development, Dorset House Publishing, 2000

(Hunt and Thomas, 2000) Hunt, Andrew and Thomas David, The Pargmatic Programmer, Addison Wesley, 2000.

(Orr, 2002) Orr, Ken, CMM versus Agile Development: Religious Wars and Software Development, Agile Project Management, Executive Report, Vol. 3, No. 7, Cutter Consortium,2002.

(Palmer and Felsing, 2002) S. Palmer and J. Felsing, A Practical Guide to Feature-Driven Development, Prentice Hall, 2002

(Pierce, 2004) Pierce, Dan, Extreme Programming Without Fear, Embedded Systems Programming, March 2004.

(Royce, 1970) Royce, Winston, Managing the Development of Large Software Systems: Concepts and Techniques. Proc. Wescon, Aug 1970.

(Rautiainen, Lassenius and Sulonen, 2002) Rautiainen Kristian, Lassenius Casper, and Sulonen Reijo, 4CC:A Framework for Managing Software Product Development, Engineering Management Journal Vol.14 No.2 June 2002, pp.27-31.

(Rautiainen, Vuornos and Lassenius, 2003) Rautiainen Kristian, Vuornos Lauri and Lassenius Casper, An Experience in Combining Flexibility and Control in a Small Company's Software Product Development Process. ISESE'03

(Schach, 2004) Scach, Stephen R., Object-Oriented and Classical Software Engineering, McGraw-Hill 2004.

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

(Vanhanen, Itkonen and Sulonen, 2003) Vanhanen Jari, Itkonen Juha and Sulonen Petteri, Improving the Interface Between Business and Product Development Using Agile Practices and the Cycles of Control Framework, Agile Development Conference 2003.

(Vanhanen, Jartti and Kähkönen, 2003) Jari Vanhanen, Jouni Jartti and Tuomo Kähkönen, Practical Experiences of Agility in the Telecom Industry, 4th International Conference on eXtreme Programming and Agile Process in Software Development, 2003.

(Ward and Mellor, 1986) Ward, Paul T. and Mellor Stephen J., Structured Development for Real-Time Systems, Prentice Hall, 1986.

(Wolf, 1994) Wolf, Wayne H., Hardware-Software Co-design of Embedded Systems, Proceedings of the IEEE, Vol. 82, No. 7, July 1994.