Agile Software Development of Embedded Systems

Version : 1.0
Date : 2005.04.08

Authors
Teodora Bozheva
Hanna Hulkko
Tuomas Ihme
Jouni Jartti
Outi Salo
Stefan Van Baelen
Andrew Wils

Status
Final

Confidentiality
Public

Agile Deliverable D.1

Agile in Embedded Software Development: State-of-the-Art Review in Literature and Practice

Abstract
This document contains a summary of agile software development and real-time embedded software development as well as a review of existing experiences on adopting agile methodologies and practices in embedded software development projects. In addition, the document contains also an analysis of the current status of agile practices and methods in the Agile ITEA consortium. The analysis is based on a questionnaire study that was conducted in January and February 2005.

ITEA
INFORMATION TECHNOLOGY
FOR EUROPEAN ADVANCEMENT

TABLE OF CONTENTS

CHANGE LOG

Vers.	Date	Author	Description
0.1	29.09.04	Hanna Hulkko	First draft created
0.2	21.10.04	Hanna Hulkko	The first version of the empirical body of evidence
0.3	10.3.2005	Tapio Matinmikko	The contributions of Teodora Bozheva, Jouni Jartti, and Andrew Wils added
0.4	1.4.2005	Tuomas Ihme,	Integration of and corrections and additions to all chapters
0.5	7.4.2005	Outi Salo	"Current Status on Agile Methods in European Software Development Organizations: A Questionnaire Study" chapter added
1.0	8.4	Tuomas Ihme	Refinements of the chapter organization and corrections of the numbering of figures and tables

APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	Identification

1. INTRODUCTION

This document contains a summary and review of the existing experiences on adopting agile methodologies or practices in embedded software development. In addition, a brief overview on the characteristics distinctive to agile software development as well as to embedded software development is presented. The document is based on a literature survey and a questionnaire study, which the existing experiences have been collected from.

The document is structured as follows: First, agile software development and real-time embedded software development are characterized. Secondly, a summary and review of the existing experiences on adopting agile methodologies or practices in embedded software development projects is presented. Thirdly, an analysis of the current status of agile practices and methods in the Agile ITEA consortium is described. Every main chapter contains also an overview or introduction section.

2. AGILE SOFTWARE DEVELOPMENT

Author: Teodora Bozheva, ESI

2.1 Overview

In the field of software development there are no practices or processes that fit all the projects. Today's enterprise solutions are complex, time critical and developed against rapidly changing business needs. Agile methods recognize these factors and instead of trying to resist them, embrace changes via business value based prioritization, short feedback cycles and quality-focused development. When appropriately used they bring a number of business benefits as better project adaptability and reaction to changes, reduced production costs, improved systems quality and increased user satisfaction with the final solution.

2.2 Home ground

B. Boehm and R. Turner [1] define five factors that organizations and projects can use to determine whether they are in either the agile or disciplined home grounds, or somewhere in between. The term “disciplined” is only used to refer to the traditional plan-driven methods without implying that the agile methods are non-disciplined. **Table 1** summarizes the home grounds:

Table 1. Agile and Disciplined Method Home Grounds

Characteristics	**Agile**	**Disciplined**
Application		
Primary Goals	Rapid value; responding to change	Predictability, stability, high assurance
Size	Smaller teams and projects	Larger teams and projects
Environment	Turbulent; high change; project-focused	Stable; low-change; project/organization focused
Management		
Customer Relations	Dedicated on-site customers; focused on prioritized increments	As-needed customer interactions; focused on contract provisions
Planning and Control	Internalized plans; qualitative control	Documented plans, quantitative control
Communications	Tacit interpersonal knowledge	Explicit documented knowledge
Technical		
Requirements	Prioritized informal stories and test cases; undergoing unforseeable change	Formalized project, capability, interface, quality, forseeable evolution requirements
Development	Simple design; short increment; refactoring assumed inexpensive	Extensive design; longer increments; refactoring assumed expensive
Test	Executable test cases define requirements, testing	Documented test plans and procedures
Personnel		
Customers	Dedicated, collocated CRACK* performers	CRACK* performers, not always collocated
Developers	At least 30% full-time experts able to tailor [and revise] a method to fit a new situation; no personnel, who, with training, is able to perform just procedural steps (e.g. coding a simple method, simple refactoring); no personel, who is not able or not willing to collaborate or follow shared methods.	50%** experts able to tailor [and revise] a method to fit a new situation; 10% throughout; 30% personnel, who, with training, is able to perform just procedural steps (e.g. coding a simple method, simple refactoring); no personel, who is not able or not willing to collaborate or follow shared methods.
Culture	Comfort and empowerment via many degrees of freedom (thriving on chaos)	Comfort and empowerment via framework of policies and procedures (thriving on order)
* Collaborative, Representative, Authorized, Committed, Knowledgable ** These numbers will particularly vary with the complexity of the application		

Since all the items in the table are self-explained, we are not going to discuss them in more details.

2.3 Approaches

There are a growing number of methods for agile software development, and a number of agile practices such as Scott Ambler's Agile Modeling [2]. The best known ones include: eXtreme Programming (XP) [3], Scrum [4, 5], Feature Driven Development (FDD) [6], Adaptive Software Development (ASD) [7], Lean Development (LD) [8], Crystal methods [9], and Dynamic Systems Development Method (DSDM) [10]. Authors of all of these approaches (except LD) participated in writing the Agile Software Development Manifesto[1], which establishes the backbone of all the agile approaches.

While individual practices are varied, they fall into six general categories[2]:

- Visioning. A good visioning practice helps assure that agile projects remain focused on key business values (for example, ASD's product visioning session).
- Project initiation. A project's overall scope, objectives, constraints, clients, risks, etc. should be briefly documented (for example, ASD's one-page project data sheet).
- Short, iterative, feature-driven, time-boxed development cycles. Exploration should be done in definitive, customer-relevant chunks (for example, FDD's feature planning).
- Constant feedback. Exploratory processes require constant feedback to stay on track (for example, Scrum's short daily meetings and XP's pair programming). Customer involvement. Focusing on business value requires constant interaction between customers and developers (for example, DSDM's facilitated workshops and ASD's customer focus groups).
- Technical excellence. Creating and maintaining a technically excellent product makes a major contribution to creating business value today and in the future (for example, XP's refactoring).

Some agile approaches focus more heavily on project management and collaboration practices (ASD, Scrum, and DSDM), while others such as XP focus on software development practices, although all the approaches touch the six key practice areas. Good introduction to the agile approaches can be found in [11] and [12].

2.4 Business rationale for using Agile method

One can find different lists of reasons why the agile methods should be practiced and what are the business benefits from applying them. Steve Hayes [13] identifies the following fundamental factors:

- Improved return on investment (RIO)
- Early detection and cancellation of failing products
- Reduced delivery schedules
- Higher quality software
- Improved control of a project
- Reduced dependence on individuals and increased flexibility

2.4.1 Improved return on investment

Hayes [13] writes, “This is the fundamental reason to use agile methods, and it's achieved in a number of different ways. In an agile project, the initial requirements are the baseline for ROI. If the project runs to completion with no changes, then the business will get the projected returns. However, by providing frequent opportunities for customer feedback, agile methods let customers steer the project incrementally, taking advantage of new insights or changed circumstances to build a better system and improve the ROI. By delivering working software early and often, agile projects also present opportunities for early deployment and provide earlier return on smaller initial investments.”

[1] www.**agilemanifesto**.org/
[2] What Is Agile Software Development?, Jim Highsmith, Cutter Consortium

2.4.2 Early cancellation of failing projects

It's a common observation that projects go well enough during the bigger part of its duration and afterwards it is delayed for months or even cancelled [13]. In such a situation the sponsors face a difficult choice: to stop it or to continue funding it hoping to get some ROI.

One of the reasons for getting in such a situation is optimistic reporting of progress on abstract intellectual tasks such as analysis and design, because it is difficult, if not impossible to estimate them correctly until the real implementation begins [13]. This means that the traditional methods suppose a hidden delay, which could be only determined when coding has started.

Steve Hayes [13] argues that "Agile projects avoid this situation by performing analysis, design and implementation in short increments, and judging progress by the delivery of working software, which gives much more solid feedback on actual vs planned progress. Overly optimistic planning becomes obvious much earlier in agile projects, giving the sponsor the chance to review the costs and benefits of the project, and where appropriate cancel the project with minimal investment."

2.4.3 Reduced delivery schedules

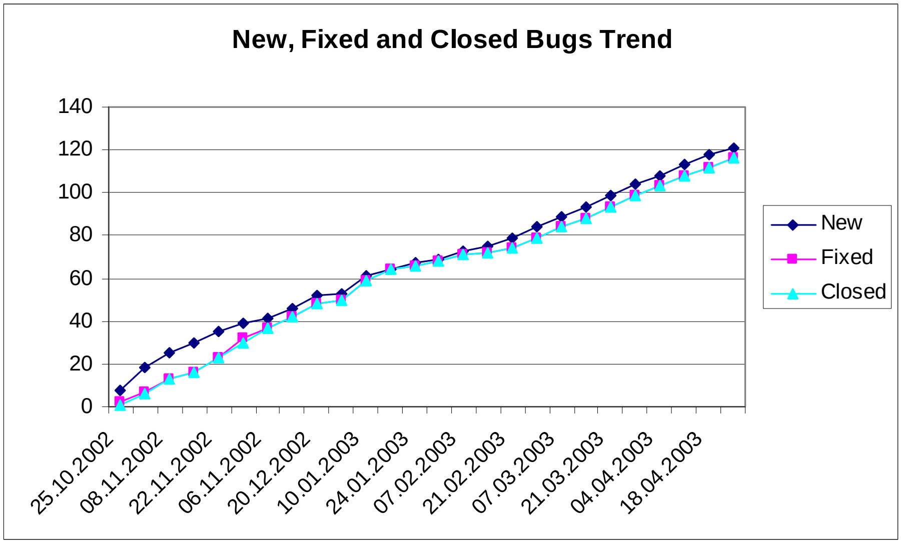
The agile methods instruct focussing on high-priority features and using short development iterations. This reduces the delivery schedules, which opens better business and market opportunities.

Developing adaptable products is an aim of most of the software organizations. By maintaining the product in a running and near shippable state, modifications to it can be quickly introduced. The ability of the team to embrass the changes required by the clients and rapidly incorporate them in the current product is a powerful mean to retain satisfied customers.

2.4.4 Higher quality

Steve Hayes [13] writes that "Of the four fundamental variables you can use to control a software development project, cost, time, scope and quality, most agile methods explicitly use scope as their control variable. All agile methods emphasise the production of high quality software, and extreme programming in particular adds a number of practices to support this objective."

The participants in the eXPERT project directly observed the higher quality produced using the XP+PSP[3] method. For instance the defects uncovering and removal followed a stable trend without accelerations as shown in Figure 1. Accelerative defect trends were typical for previous projects.



[3] A combination of XP with some PSP (Personal Software Process by W. Humphrey) practices

Figure 1. Bugs tend applying the eXPERT mehod[4]

2.4.5 Improved control

Agile projects typically produce fewer paper artefacts. However, at the end of every iteration (which is up to two months long) they deliver a running product, which gives sponsors and customers improved visibility and control of a project. Hayes [13] argues, “This is further enhanced by emphasis on integrated, multi-disciplinary teams, where information is routinely shared, and highly visible.”

2.4.6 Reduced dependence on individuals and increased flexibility

Hayes [13] writes, “Agile projects emphasize sharing of information, and performing analysis, design and coding on a team, rather than individual basis. This helps prevent situations where development is critically dependent on an individual, who might become unavailable on short notice. It also means that development doesn't become bottlenecked. On an agile project it should be possible to get everyone on the team working in the same area of the system, and change this area from one week to another depending on business priorities.”

To wrap up, the agile methods do deliver direct benefits to business and for projects in turbulent environments, the flexibility and feedback provided by these practices may be particularly critical.

2.4.7 Statistics

During November 2002 to January 2003, Shine Technologies ran a web-based survey [14] to gauge the market interest in Agile Methodologies. The survey consisted of 10 questions, and received 131 valid submissions. Figures 2 to 6 depict the results of the following five questions:

Question 4: Has adoption of Agile processes altered the quality of your applications?
Question 5: Has adoption of Agile processes altered the cost of development?
Question 6: Has adoption of Agile processes altered the level of business satisfaction with the software?
Question 7: What feature of your Agile processes do you like the most?
Question 10: What proportion of projects do you believe are appropriate for Agile processes?

The survey [14] comments Figure 2 that “Adoption of Agile processes has had a significant effect on the quality of applications delivered. Of knowledgeable respondents, 88% claimed better or significantly better quality. Across all respondents this number falls to 84%. Only 1% of knowledgeable respondents believed that quality was adversely affected in any way.”

[4] eXPERT project (www.esi.es), Nemetschek’s case study

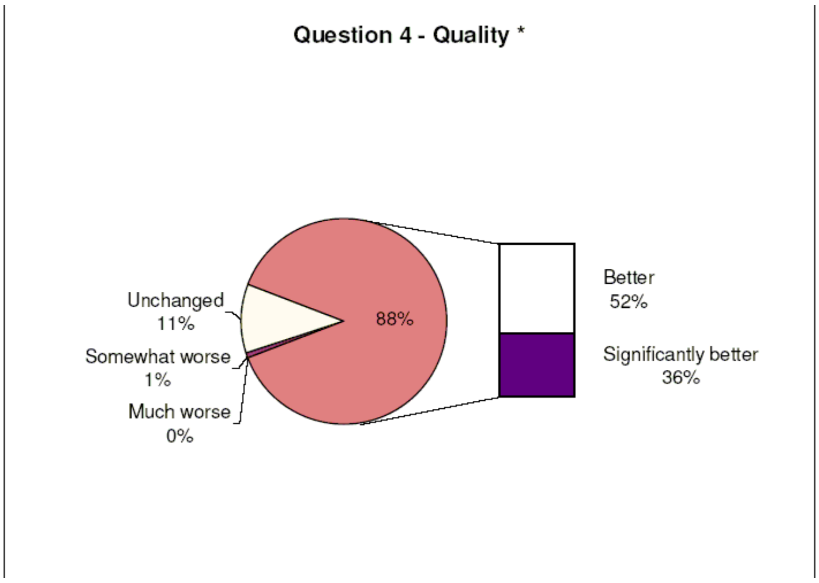


Figure 2. The results from the question "Has the adoption of Agile processes altered the quality of applications?" [14]

The survey [14] summarizes Figure 3, "Across respondents with average knowledge or better, 48.6% believed that development costs were reduced. Including the responses that indicated that costs were unchanged, a whopping 95% believe Agile processes have either no effect or a cost reduction effect."

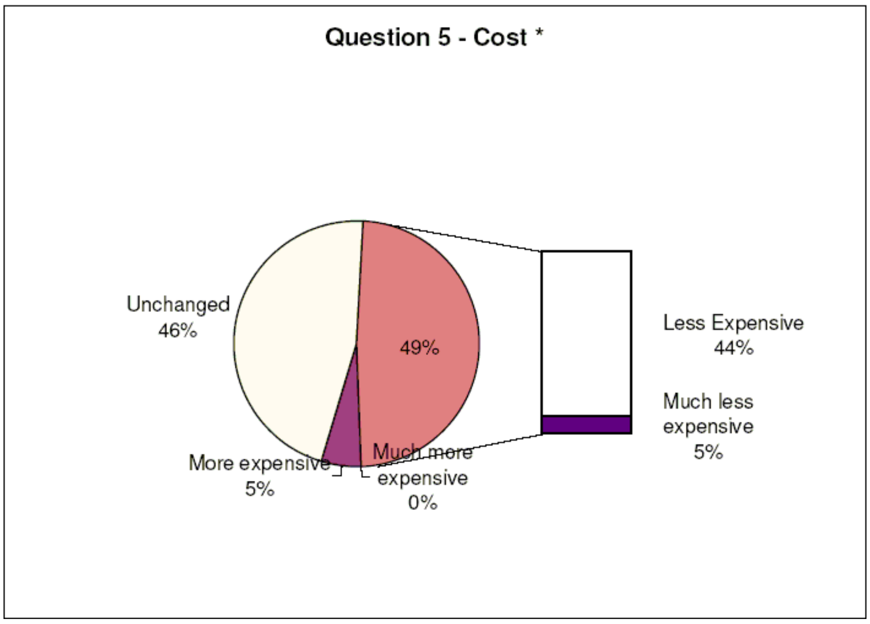


Figure 3. The results from the question "Has adoption of Agile processes altered the cost of development?" [14]

Figure 4 represents that business satisfaction of better or significantly better was a phenomenal for 83% of respondents with average knowledge or better. Only 1% of the respondents believe it has had a negative effect.

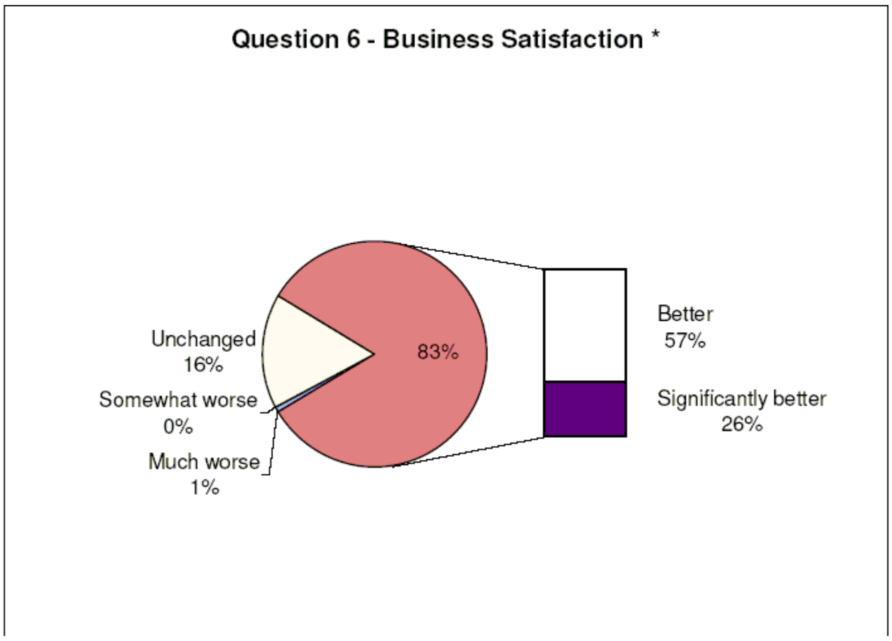


Figure 4. The results from the question "Has adoption of Agile processes altered the level of business satisfaction with the software?" [14]

Respondents ranked "Respond to change over plan" (47.3%) and "People over processes" (30.5%) as the most positive features of Agile processes (Figure 5). This appreciation of a responsive and people-centric model is a striking change from the traditional methodologies that value plans and processes.

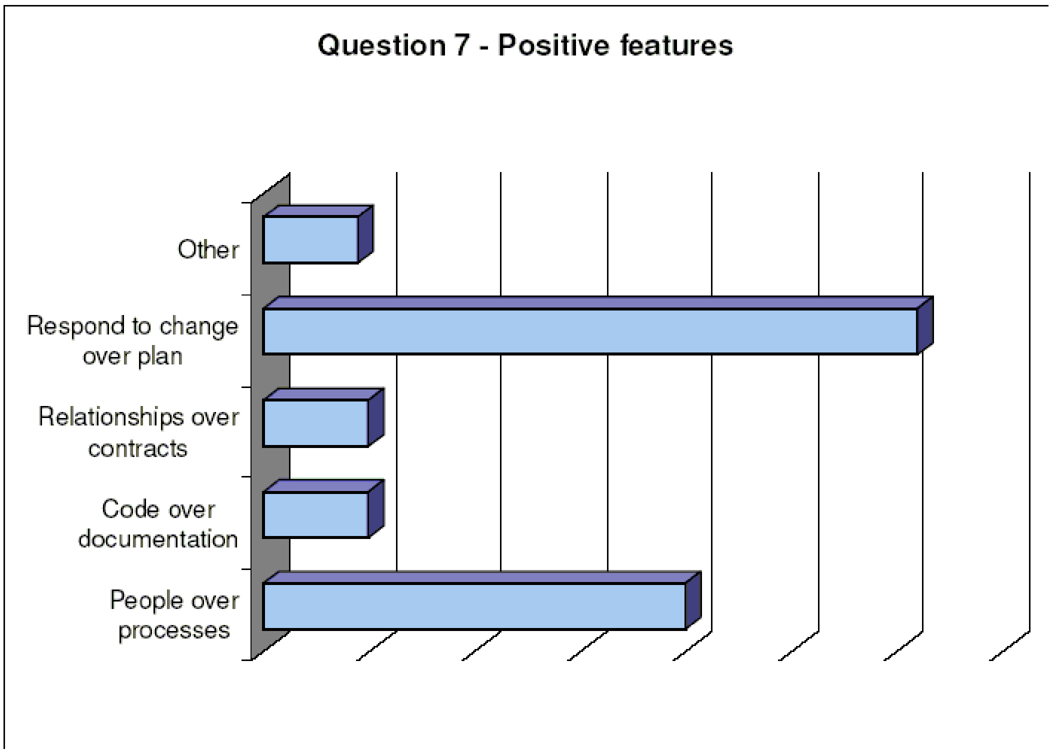


Figure 5. The results from the question "What feature of your Agile processes do you like the most?" [14]

The survey [14] summarizes Figure 6, "Only 16% of respondents believe that Agile processes are applicable to all projects. This is in line with the Agile belief that it should be applied only where it will deliver benefit. Interestingly 88.5% of respondents believe that Agile processes should be used at least half the time. This indicates that Agile processes should be used only for the right projects, and that there is room for other methodologies to sit along side Agile and be used on a project-by-project basis as appropriate."

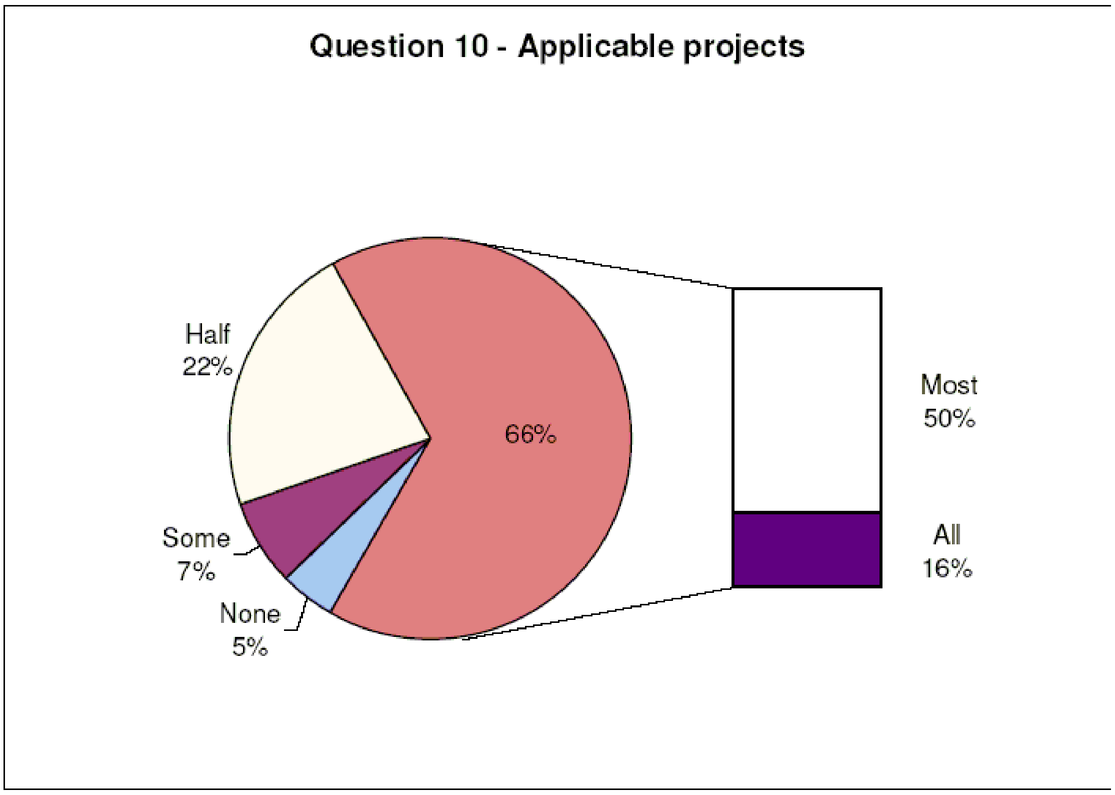


Figure 6. The results from the question "What proportion of projects are appropriate for Agile processes?" [14]

3. DEFINITION OF AGILE SOFTWARE DEVELOPMENT

Author: Jouni Jartti, Nokia

Agility in embedded SW development is an ability to make changes in order to maximize the overall customer value whilst minimizing the loss of initial customer value; customer value being the cross function of money, time, effort, functionality or any quality characteristics like reliability or dependability.

Figure 7 clarifies the benefits of agile SW development. The goal is to increase customer value through change driven Agile SW development. Change has always cost and benefits, it is important to minimize the cost of the change, and try to maximize the benefits in the new improved customer value situation.

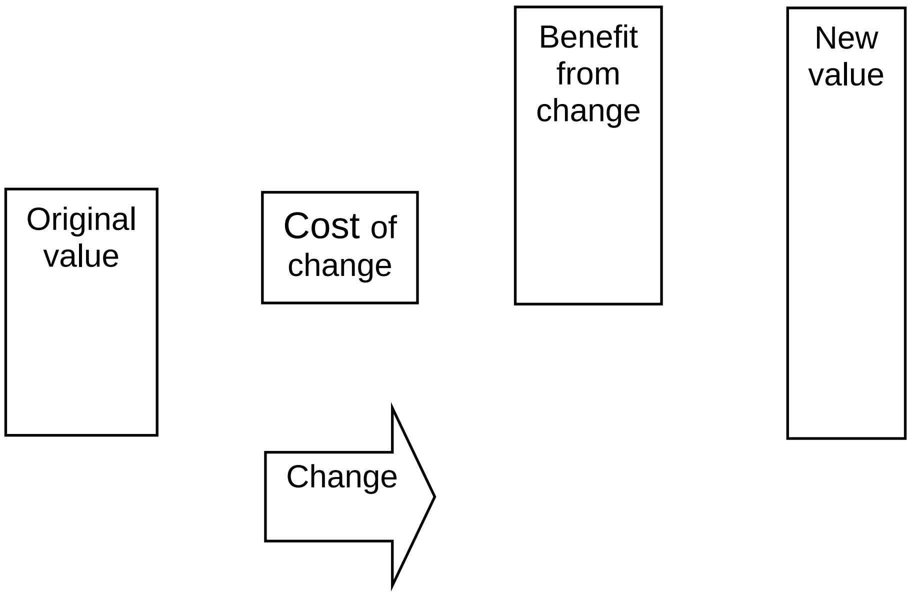


Figure 7. The benefits of agile SW development

The word agility means the quality of being agile; readiness for motion; nimbleness activity, dexterity in motion. Agility can be also seen as an attribute of an organization and its processes. Agility is defined as the ability to react to changing situations quickly, appropriately, and effectively. In other words, an agile organization notices relevant changes early, initiates action promptly, creates a feasible and effective alternative plan quickly, and reorients work and resources according to this new plan quickly and effectively.

Software development agility has been identified as a necessary, new approach to software development in domains where new business concepts, new customer segments, new products, and technologies need to be introduced and used under tight time pressures. Several approaches have been proposed to support process agility, such as Extreme Programming [3]and RUP [15]. The principles of agile software process have been phrased as follows:

- "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. Working software is the primary measure of progress. "

- "Continuous attention to technical excellence and good design enhances agility. Simplicity--the art of maximizing the amount of work not done--is essential. "

The embedded SW development has new challences for agile SW development. Hardware sets tight requirements for the software and requirements for hardware related software cannot be frozen before development starts. Also in the embedded systems performance and reliability are key issues not just functionality and therefore co-design is needed between hardware and software. Agile SW development should welcome changes also in these demanding embedded environments and produce working systems fast.

4. REAL-TIME EMBEDDED SOFTWARE DEVELOPMENT

Authors: Andrew Wils, K.U.Leuven
Stefan Van Baelen, K.U. Leuven
Hanna Hulkko, VTT
Tuomas Ihme, VTT

4.1 Overview

Embedded software development is characterized by the existence of resource constraints. Dependent on the application domain, this limited availability of resources can be of a different nature: processor capacity, RAM, storage memory, bandwidth, power, user interface, etc. Methodologies for Embedded Software Development typically provide support for developing software that has to perform within such a resource-constraint environment.

Due to the real life usage characteristics of embedded systems, e.g. need for accurate real-time behaviour, need for reliable operation, and long lifecycle of products, embedded software as a part of an embedded system has to often meet several stringent and specific requirements. Common software development processes often do not suffice to ensure and certify these requirements. This chapter provides an overview of typical embedded software characteristics and development methodologies.

4.2 Methodologies for High-Level Embedded Software Development

4.2.1 Overview

Traditional spiral, waterfall and V models for embedded software development have evolved into more elaborate processes and methodologies, suitable for more complex embedded systems. As component-based software design is especially suited for embedded systems development these processes are often based on traditional component-based methodologies.

Examples of component and object-oriented methodologies include KobrA [16], COSM [17], Catalysis [18], Octopus [19], ROOM [20], the Rose RealTime Approach [21], RT Perspective method of ARTISAN [22], ROPES [23], and Telelogic Tau [24]. Complementing these are more standards-driven processes such as CMM/CMMI and ISO. In most methodologies, UML takes a paramount role as the de facto modelling standard. The UML Profile for Scheduling, Performance and Time, the UML Profile for Quality of Service and the newly released UML 2 specification further increase the suitability of UML for embedded and real-time systems [25].

However, software engineering research and products have had only a limited success in addressing the more constrained traditional problems of real-time and embedded systems [26].

4.2.2 Unified Process

The Unified Process (UP) [15] is a component-based process that is use-case driven, architecture-centric, iterative and incremental. It describes a proven family of techniques that supports the complete software development life cycle. The UP was brought together by the people behind the UML and makes heavy use of it. The fundamental idea of the UP is to retain the organization of the spiral model (that prescribes to plan a phase, analyse risk, and do the step) in an iterative process where every macro-phase is organized in an incremental way, by means of several iterations (Figure 8).

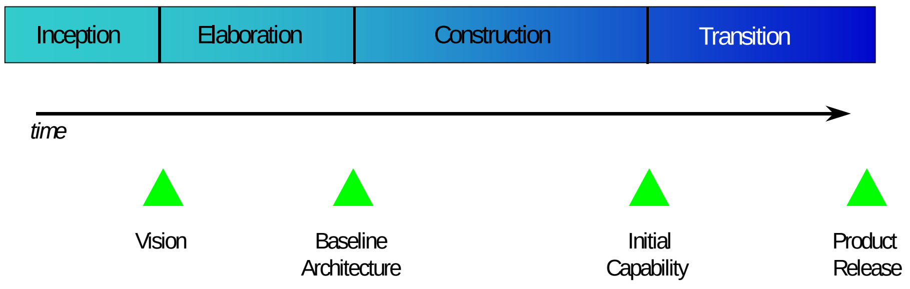


Figure 8. The Unified Process is an iterative and incremental process.

4.2.3 Dess

The ITEA DESS project defines a methodology for component-based development on top of the UP [27], and a notation for component modelling on top of UML. The DESS methodology captures the best practices of the classical V development model and incremental development approaches. It defines 3 workflow V's that can logically be connected by means of artefact flows but without any time ordering. The 3 Workflow-V's, which can be seen as standing in parallel, are:

- A Realization Workflow-V, containing workflows dealing with the realization of a system (system development);
- A Validation & Verification Workflow-V, containing workflows dealing with validation, verification and testing;
- A Requirements Management Workflow-V, containing workflows dealing with establishing and maintaining the requirements of the system on all levels of the organization.

Although the approach supports incremental development, only the Requirements Management Workflow-V partially deals with evolution during the system lifecycle. But the DESS methodology can be used as a starting base for component-based development, extending it with adequate methodological support for evolution. The DESS component notation defines a component specification and documentation standard on top of UML. It supports component interface specification, component connections (wiring), hierarchical component development, component frameworks and connector specification and design. Although versioning is considered as a part of the component identification, little support is present for component and interface evolution. Also the DESS component notation can be used as a starting base for component modelling, extending the notation and specification with adequate notational support for evolution.

4.2.4 Empress

The ITEA EMPRESS project positions evolution of real-time and embedded component systems in the UP, allowing it to cope with product families and software evolution at design-time as well as run-time. The resulting EMPRESS process has an extra run-time phase, next to the existing Inception, Elaboration, Construction and Transition phases. Also, it extends existing UP workflow details, activities and activity steps [28].

4.2.5 MDA

A recent trend in embedded software development is the application of model transformations. The Model-Driven Architecture (MDA) [29-31] as defined by the Object Management Group (OMG) is a state-of-the-art architecture for model-driven software development. MDA wants to obtain a maximal software reuse by abstracting the software development from the actual platform to run upon and the implementation language to use. This is achieved by developing a software model on a higher, platform-independent level, which is later on transformed into a model on a lower, platform-specific level, taking into account all specificities of the target platform.

The key principle of MDA is the separation of a software system into a number of parts (models), such as a specification of the system functionality and the specification of the functionality realization (implementation) on a specific platform. MDA provides support for the realization of a system on several concrete platforms, both in parallel to each other as being part of the portability support during the total lifetime of the system. The architectural framework defined by MDA offers a number of guidelines to structure the models and transform higher-level models into lower-level ones. These mappings are used to target the described software system towards a concrete language and execution environment. As such, concepts for control software can be developed uniformly, and afterwards tailored to a concrete machine and execution environment.

System integration is realized in MDA by integrating subsystems on the model level, supporting the reusability, integration and interoperability of the system concepts and the evolution of the software on top of the fluctuations of the state-of-the-art platform technologies. The basic concepts of MDA are models and model mappings. A model is a formal specification of the functional logic, the structure and the behaviour of a system at a certain abstraction level. Abstraction levels introduced by MDA are as follows:

- The Computation Independent Model (CIM) is a system model that provides an abstract view on the software components within the system. It is computation independent because it doesn't further describe the design details of the components and their internal functioning. The CIM specification is thus limited to a description of the goals and functions of the software system, not how they operate or how they are realized.
- The Platform-Independent Model (PIM) details the computational algorithms and the design details of the various software components, and further specifies the interaction between these components. But the model still hides the technical details and realization of the system (language, middleware platform, execution environment, operating system). The algorithms and component interactions are described on a platform independent manner, abstracting the technological realization details that are irrelevant for the fundamental functional realization of the system and its components.
- The Platform-Specific Model (PSM) specifies the realization of the functionality on a specific platform. Technological details will be introduced related to the target platform. This model for instance uses language-specific concepts and concepts from the operating system on which the embedded software will run.

The MDA approach is very promising for embedded software development, because the PIM to PSM mapping of MDA provides support for the realization of the same system on several concrete platforms. The realizations can be done both in parallel to each other at one moment in time, or as a retargeting of the system towards a different hardware and/or software environment as part of an evolution or porting process.

4.3 Characteristics of Real-time Embedded Software

Software development for Real-time Embedded (RTE) systems is characterized foremost by special requirements that are asked from the software. A thorough characterization of real-time embedded software characteristics has been done in a previous ITEA project DESS. The following description is a summary of the DESS results [32] with some additions by the authors of this deliverable.

4.3.1 Embedded system

4.3.1.1 Common characteristics

Embedded systems are all dedicated to one or more specific tasks. Large networked systems can include the whole technological spectrum of embedded systems from deeply embedded application specific hard real-time systems to software intensive applications.

4.3.1.2 Generic definition

Embedded systems are systems that are embedded as components in a larger system [26].

4.3.2 Embedded software

4.3.2.1 Common characteristics

Embedded software is often constrained by the hardware's capabilities, especially by the amount of memory and processing power available [33]. Because many embedded systems rely on battery power, it is crucial to design and validate them for controlled power consumption. Also, since embedded software is often implemented on small machines with limited RAM, it is necessary to include memory considerations in the design. [34] In addition, the hardware constraints can also hinder development by e.g. preventing running all test code at the same time in the test environment [35].

4.3.2.2 Generic definition

According to IEEE [36], the part of an embedded system implemented with software, i.e. embedded software, can be defined as "software that is a part of a larger system and performs some of the requirements of that system; for example, software used in an aircraft or rapid transit system". Embedded software consists of built-in computer programs for controlling high value-added products such as switching and production control systems, space instruments, wireless communication devices, home electronics goods, and mechatronic machines [37]. It can be classified based on its purpose of use to at least three major classes [37]:

- System software, including the operating system, communication, device control and platform-specific functions (e.g. initialization tests, memory loader etc. by which it is possible to deal with hardware problems)
- Product family software not unique to the application (to be used in products of several kinds)
- Application software, which is specific to one application

4.3.3 Real-Time System

4.3.3.1 Common characteristics

The real-time aspect of the embedded systems becomes apparent in many ways. Depending on the type of application, emphasis is put on delivering a service, the quality of the service, or a certain speed in responsiveness in delivering the service.

4.3.3.2 Generic definition

The correctness of real-time systems depends not only on the logical results but also on the time at which the results are produced [38].Usually, three types of "real-time" are distinguished:

- Hard real-time. The deadlines of hard real-time services **must** be met because their missing can cause severe consequences. In the case of missing a hard real-time deadline, the embedded application will be considered "failed".

- Soft real-time. Soft real-time constraints allow that deadlines are missed, however in this case the quality of service comes into play. If the quality might be endangered in such a way that the overall result becomes unacceptable, the system may be considered "failed" as well.
- Statistical real-time. Statistical real-time deadlines may be missed, as long as they are compensated by faster performance elsewhere to ensure that the average performance meets a hard real-time constraint. To be able to fully assess the consequences of the statistical behaviour, stochastic analysis is required. However, it is always possible to transform this into a deterministic analysis by investigating the worst case situation.

4.3.3.3 Metrics

Test equipment will have to be prepared to allow verification of several *timing constraints* (Figure 9):

- *Period*: As the controller behaves periodically, all necessary computations and adjustments of actuators must be completed within the given time interval. The period is e.g. imposed by a sensor device delivering its value periodically.
- *Deadline*: The deadline of a task is measured relative to the beginning of the period. The task has to be guaranteed to have finished before this deadline. Note that the deadline is less than the period length, if there are several tasks that have to be completed within the given period.
- *Response Time*: This is the longest time ever taken by a task from the beginning of its period until it completes its required computation, i.e. including all possible interferences by higher priority tasks and interrupt routines. The real-time constraint is represented by the fact that the worst-case response time of a task has to be smaller than its deadline.
- *Release Time*: An offset value that represents the arrival (release) of a certain task with regard to the beginning of the period. The release time of a task must not be later than thedeadline of the task minus WCET, where WCET is the Worst-Case Execution Time of the task.
- *Jitter*: The amount of time the response time of a task can vary due to interferences and inaccuracies caused to it and its predecessors.

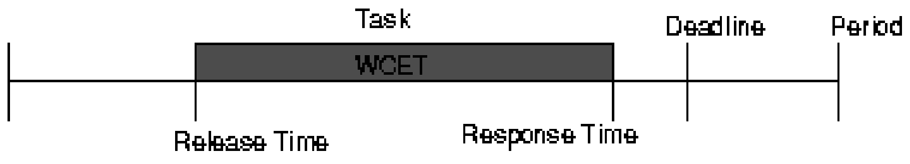


Figure 9. **The definition of time-constraint metrics**

4.3.4 Memory Constraints

4.3.4.1 Common characteristics

Memory constraints come in two flavours: constraints on the size and on the behaviour. Size constraints are particularly important in products for mass consumer markets. Indeed, if the memory of such products has to be increased, then this will have an immediate influence on the price of the appliance. Even for products where price is not an issue, the size is best determined in such a way that ample space is left for possible future extensions. Especially when microcontrollers with on-board RAM and/or ROM are used, a substantial gain in cost can be obtained by designing the software in such a way that it fits the on-chip resources, and no additional, external memory is required.

If there is a requirement for predictable, deterministic behaviour of the memory, then this constraint has an influence on the way the memory is used. Again, there is a hard and a soft version of the constraint. The hard variety demands that all bytes that are used are somehow accounted for by the application. For the soft variety, this is not required. However, deterministic behaviour usually disallows dynamic memory use, as well as time-unpredictable features such as virtual memory, cache or garbage-collection schemes.

4.3.4.2 Generic definition

Memory constraints can be summarised as follows:

- Cost: minimise the size to keep it as low as possible. For microcontrollers, the cost function has a discontinuity when the size of the required memory resources exceeds what is available on-chip.
- Load: maximise the size to cater for future extensions
- Hard determinism: no dynamic behaviour, and all bytes accounted for
- Soft determinism: no dynamic behaviour.

4.3.4.3 Metrics

The following metrics apply to memory constraints (and are quite self-explanatory):

- Size
- Load
- Cost
- Dynamic behaviour
- Accounting for the data

4.3.5 CPU constraints

4.3.5.1 Common characteristics

Two main issues are important to CPU constraints:

- The CPU performance must be well dimensioned. The CPU must be sufficiently powerful so that the application's algorithms still satisfy the real-time constraints. Also, the CPU must not be used at a 100% of its capacity at this time, to allow for future expansions. If the application is intended for mass markets, then care must be taken not to go for a performance overkill solution, because that might not be cost effective.
- The CPU usage must be deterministic. To ensure a deterministic CPU usage, it is better to take a time-triggered approach rather than using an interrupt-driven approach, although this can not always be avoided.

4.3.5.2 Generic definition

CPU constraints can be reduced to:

- Overall performance must be sufficient to support current requirements
- Load must allow to cater for future extensions
- Determinism: no dynamic behaviour, or at least try to reduce to the strict minimum

4.3.5.3 Metrics

The following characteristics of the CPU can be measured (e.g. with a logic analyser) or estimated by code inspection and processor spec study):

- Intrinsic performance (from datasheets)
- Actual performance (to be measured with e.g. logic state analyser)
- Load (to be measured with e.g. logic state analyser)
- Determinism of the behaviour (to be checked with e.g. logic state analyser and by inspection of the code)

4.3.6 Bandwidth Constraints

4.3.6.1 Common characteristics

A number of embedded applications have very stringent bandwidth requirements, dictated by the amount of communication that is required (e.g. over buses). It specifies the maximum amount of data that can be moved over a channel, e.g. in bits per second. Furthermore, there may be constraints on the total time that a communication is allowed to take.

4.3.6.2 Generic definition

The two constraints regarding bandwidth are:

- Maximum transmission speed, which determines the maximum bandwidth the signal can/should use
- Total transmission time, which becomes important if the speed at which the data is sent, has to be artificially reduced to meet the previous constraint, either due to the amount of data or the unavailability of the full bandwidth.

4.3.6.3 Metrics

Both constraints can be measured:

- The average, maximum and minimum amount of bits, sent per second, can be monitored
- The total time of a typical, a best case and a worst-case transmission can be measured to evaluate the quality of service rendered.

4.3.7 Power Consumption Constraints

4.3.7.1 Common characteristics

Especially in handheld appliances, power management has an important impact on the autonomy of the system. Therefore, it must be under control right from the beginning of the design. Also, as the power is in such cases drawn from a battery, these constraints will have an impact on the overall design of the appliance, and indeed, the choice of battery. Specifically for software, care must be taken that the hardware is used only when required (e.g. if the system contains a heater or a backlight, don’t switch it on when it is not necessary).

4.3.7.2 Generic definition

There are several direct and derived aspects to the power consumption constraints, and this time, not all of them can be expressed as electric or temporal quantities:

- Autonomy of the system
- Cost
- Weight
- Dimensions

4.3.7.3 Metrics

The power consumption of the embedded system can readily be measured:

- Power = voltage * current, both of which can be measured
- Energy capacity of the battery (from the battery specs)

Other quantities:

- Dimensions
- Weight
- Cost

4.3.8 Functional characteristics

Functional characteristics describe the system. Although they can be categorised, deriving metrics from them is quite impossible. This feature therefore has been omitted

4.3.9 Communication

4.3.9.1 Common characteristics

Embedded systems usually have one or several means to communicate besides the User Interface. They can be classified as follows:

- Bus as an internal communication between its constituent parts
- Bus as a dedicated external communication to similar and/or different devices
- Network as an open connection to similar and/or different devices
- Physical read/write devices (magnetic/optical media, …)

All possibilities share one feature: no matter how the connection(s) is (are) ultimately established, strict protocols are to be followed. Those protocols dictate some of the real-time and performance requirements.

4.3.9.2 Generic definition

A distinction can be made based on the communication type:

- Internal bus
- External bus
- (Wireless) network connection.
- Carrier

They all make use of one or more protocols.

4.3.10 User interface

4.3.10.1 Common characteristics

Most embedded systems perform some interaction with the human operator as well. To accomplish this, a large variety of interfaces are available:

- Keyboard
- Display with GUI
- touch screen
- voice input
- voice output
- image input
- fingerprint (for identification and authentication)
- sensors, measuring other types of data

4.3.10.2 Generic definition

A classification is needed here to describe the different I/O possibilities:

- Manual input (keyboard, touch screen)
- Visual input (scanners)
- Visual output (display, possibly with GUI)
- Voice input
- Voice output

- Sensor input

4.3.11 Command and control

4.3.11.1 Common characteristics

Command and Control functions usually remain restricted to the embedded system itself, but in some applications, an embedded system may get additional authority to control other (embedded) systems of the larger system it is part of. This is usually done by allowing the embedded system to take in, or drive discrete and/or analog lines.

4.3.11.2 Generic definition

Basically, only two kinds are available:

- Discrete I/O
- Analog I/O

4.3.12 Operational characteristics

For these types of characteristics, it is oftentimes quite difficult to distil some generic definitions. Even more cumbersome (and indeed in many cases impossible) is to derive metrics from them. Therefore, definitions and/or metrics were only added where they made sense. In the other cases, they were simply omitted. Only extensive testing and verification against the defined requirements for the system can reveal whether a design goal is not met. Needless to say that in view of the large variety of systems, this task is very difficult to generalise.

4.4 Quality Attributes of Embedded Software

4.4.1 Quality of Service

4.4.1.1 Common characteristics

For most of the partners, this is the number one characteristic. Although “Quality of Service” is a very vague term, it can be described by the following three phrases:

- The service must be delivered with the desired functionality;
- The service must be delivered in a timely manner;
- Sometimes, also a quick readiness after switching the appliance on is very desirable.

The level of fulfillment of these aspects can vary even for one and the same embedded system, depending on its set of “missions”, or intended purposes. On one occasion, only a very basic functionality/service may be required, while put in another situation, much more is expected from the same appliance. Depending on the requirements put to it, it may happen that the additional functionality/services required from the system need not to be delivered at the same high quality of the basic functionality, and that it is perfectly acceptable that the quality goes down even further when the system is solicited even more. This is usually called “graceful degradation”.

Although “Quality of Service” is impossible to quantify, it can be broken down into other aspects, which may prove a little easier to handle, e.g.:

- Functionality (see Section 4.3.8): can be checked against specifications
- Timing constraints (See section 4.3.3): can be quantified
- Dependability (see Section 4.4.2)

4.4.2 Dependability

4.4.2.1 Common characteristics

This characteristic seems very subjective at first. Yet, it is of utmost importance for most embedded systems. The Quality of Service provided by e.g. a handheld appliance “depends” on it; In case the embedded system is used in a safety-critical situation, the safety of the operators (and possibly many more people) “depends” on it.

Operators must be able to put their trust into the system. This can only be achieved when the system has a high rate of availability, and provides reliable service to the user. However, this is usually not enough. How can an operator put his trust in a system when there is no way of verifying the correctness of its operation? The notion of safety comes into play at this point. This means that dependability can be further expressed in terms of these three characteristics, which will in turn provide a way to quantify dependability.

When redundancy is built into the system, then this will have an impact on the dependability.

4.4.2.2 Generic definition

Trustworthiness of a system such that reliance can justifiably be placed on the service it delivers. Availability, reliability and safety must be ensured.

4.4.3 Availability

4.4.3.1 Common characteristics

In applications where short periods of downtime are acceptable, they must be minimised in order to maximise the availability of the service that is delivered (closely related to the Quality of Service). A number of statistical methods, based on the history of the system, have proven their value on the hardware level. For software, however, most of the stochastic approach is still being investigated and/or developed. Dependent on how redundancy is built into the system, there will be an impact on the availability. See section 4.4.4 for a discussion in combination with related characteristics.

4.4.3.2 Generic definition

Measure of correct service delivery with respect to the alternation of correct or incorrect service (dependability with respect to readiness for usage).

4.4.3.3 Metrics

Cumulative downtime or uptime over an extended period of time (possibly expressed as a percentage) are the typical measures used to describe this aspect. Also the number of failures per (extended) period of time (e.g. a year) can provide valuable information, or even better, the mean time of occurrence of the first failure. These are statistical metrics, which can be drawn from historic data, if available.

At this time, it makes sense to distinguish between repairable systems (either subject to maintenance and/or failing systems can be replaced entirely) and non-repairable systems (inaccessible systems like satellites). For the latter ones, cumulative up or downtime has little or no meaning. If a failure causes the system to go down, then it becomes practically impossible to get it operational again. It is not always necessary to consider the fact if a system is repairable or not, but whenever it is believed to have an impact on the characteristic, it will be taken into account.

4.4.4 Reliability

4.4.4.1 Common characteristics

The reliability is closely related to the safety: it is the length of time the system must be able to operate without the safety aspects being jeopardised. Very often, reliability and safety are therefore treated together, and usually, a trade-off will have to be made between them. A means to change reliability/safety can be adding redundant systems, or adding components to verify the correct operation of the basic functionality. The net result of redundancy is that safety increases (more error situations can be detected and trapped/reported), but that reliability decreases (the redundant system itself can fail as well).

Again, probabilistic approaches can be used to refine the reliability measure. Such a metric could be the probability that a failure occurs before a certain time elapsed. Those approaches are again much better defined for hardware than for software, and if it is not possible to derive this from historic data, an a-priori analysis is very difficult for a software application.

If redundancy is added to the system, with a decision-making algorithm to sort out the failing subsystems and (possibly) stop the whole operation as soon as the discrepancies become too big, then this will have a definite influence on the four characteristics which are closely linked together: dependability, availability, safety and reliability. In general:
- safety will go up (there are backup subsystems that can take over)
- reliability will go *down*: (there are more possible causes for failures, also the decision-making subsystem itself can fail)
- if the decision-making algorithm has the authority to shut down the overall system, then availability might go down as well (more possible failures can cause more alarming situations than before)
- the dependability will definitely be influenced by the characteristics above, but the question if it will improve or deteriorate depends on which of the three characteristics are focused.

4.4.4.2 Generic definition

Measure of continuous correct service delivery (dependability with respect to continuity of service).

4.4.4.3 Metrics

Usually, quantities like MTBF (Mean Time Between Failures) are calculated/estimated. This metric loses its meaning for non-repairable systems. The first failure is usually fatal. Therefore, it is more important to use MTTF (Mean Time To Failure) in this case.

4.4.5 Safety

4.4.5.1 Common characteristics

A more stringent aspect than reliability is the safety of the embedded application. Software safety is concerned with the problems of building software for embedded systems where failures can result in loss of life or property. Potential hazards must be defined, and the probability of their occurrence estimated. Of course, this occurrence must be avoided at all cost, but the probability is never zero, because electronic failure rates are not zero either. In this case, possible software failures must be:
- Traced, so that can not lead to one of the defined hazards (fault tree analysis);
- If it does lead to a defined hazard, the probability is lower than the one defined;
- Any occurrence is detected and announced to the operator, so that corrective actions are still possible.

Safety will definitely be influenced by redundancy. See section 4.4.4 for a discussion in combination with related characteristics.

4.4.5.2 Generic definition

Measure of continuous delivery of either correct service or incorrect service after benign failure (dependability with respect to the non-occurrence of catastrophic failures).

4.4.5.3 Metrics

Fault analysis will lead to requirements, which need to be added to the specification requirements and they must be tested against. This metric as such is unavoidably following from a stochastic analysis. If historic data is available, then a metric similar to MTBF can be redefined to reflect the Mean Time Between Catastrophic Failures, or the mean time of occurrence of the first catastrophic failure can be derived. For non-repairable systems, the MTTF should again be used instead.

4.4.6 Robustness

4.4.6.1 Common characteristics

This depends on the kind of application, but embedded systems are often used in environments where reset and/or repair are not desirable. In that case, the system must be up at all times, and it must remain functional, even when the data sent to it is erroneous. The embedded system must be able to handle any circumstance in which the overall system is used.

4.4.6.2 Generic definition

The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions.

4.4.7 Testability

4.4.7.1 Common characteristics

There are two distinct aspects to the testability of an embedded system: the testability of the application during development and/or qualification for use, and the ability of the embedded system to evaluate its own condition.

Testability at development time is not an easy issue. Some formal methods/tools allow to test and verify (and in some cases: prove) that a certain automaton is behaving the way it was designed, or conversely, that it is not. A clear advantage of this approach is that a large part of the embedded software application can be tested without the need for the final hardware. The development environment can already be used. Coverage analysis is a good example of what can be accomplished in a simulated mode.

One of the topics described in the safety aspect was the ability to detect and announce the occurrence of a failure. This means that substantial efforts must be made to allow for the implementation of Built-In-Tests, which monitor the health of the system. In order to accomplish this task, it may be required that special components are added to provide the required feedback. Although the additional components will reduce the reliability, they will have a positive effect on safety. Special care must be taken when such components, or even parts of the application, are provided by a third party.

4.4.7.2 *Metrics*

Although it is very difficult to measure the complexity and testability of software, a number of quantities can be calculated (e.g. McCabe number). Many software development tools allow the programmer to calculate this indicator. It is to be used with care though, because it is not that difficult to influence the number by non-essential tampering with the source code.

A good indicator that can be used, however, is the amount of coverage of the requirements that can be obtained from the tests that have been conducted. Tools and formal methods exist for accomplishing this task.

Defining a metric about “how well” an embedded system is capable of self-diagnosing its current state is next to impossible at this point.

4.4.8 Maintainability/Serviceability

4.4.8.1 *Common characteristics*

For some applications, it is a must that defective systems can be replaced and/or repaired very quickly. Even when this aspect is not a requirement, it is highly desirable to ensure customer satisfaction.

4.4.8.2 *Metrics*

MTTR stands for Mean Time To Repair and is an important indicator for repairable systems.
Although this is a very deterministic measurement, there are so many external factors that can influence the actual repair time. In case a more sophisticated measure is needed, a statistical approach may be taken, and instead of looking at the (constant) MTTR, the probability that the actual repair time exceeds the MTTR can be investigated. In most practical situations, however, the MTTR proves to be adequate to describe a system’s maintainability. It is clear that MTTR only applies to repairable systems.

4.4.9 Security

4.4.9.1 *Common characteristics*

Some embedded systems have direct access to public networks. The problem at hand is that the opposite also holds: the public network has access to the system. This must be handled with extreme care, and the usual solutions which hold for “standard” computers, are not powerful enough to ensure secured access.

4.4.9.2 *Generic definition*

Dependability with respect to the prevention of unauthorized access and/or handling of information.

4.4.10 Field loadable Software

4.4.10.1 *Common characteristics*

Upgrading software is a very important aspect: it allows to correct possible flaws, and/or provides a possibility to enhance the functionality in the field. Care must be taken that the new software versions are still compatible with the older hardware.

4.4.10.2 Metrics

Some measures will have to be taken and verified to ensure data integrity during and after download. Needs proof by analysis.

4.4.11 Configurable software

4.4.11.1 Common characteristics

From a supplier's point of view, it is very interesting to reuse large parts of previous systems. -New products are often based on software components or products baselines developed in the previous projects [39]. Thus, considering the issues related to the reusability of the product under development is necessary. Still, different customers will have different requirements, even when the same standards are followed. Also, if the same software is to be used on different systems with different capacity, then developing the software in such a way that it is scalable is an important advantage. Control over the scale itself is then exerted through configuration. Making the software configurable is a major factor in decreasing development time.

4.4.11.2 Metrics

Care must be taken that only the parts of the software that apply to the application at hand are activated and nothing else. Needs proof by analysis

4.4.12 Flexible Software

4.4.12.1 Common characteristics

Closely related to the previous topic is the flexibility of the software that is developed. Not only is it desirable to make small adjustments to a particular application (configurable), it also has to possible to accommodate for late changes in specs, or simply reuse of a certain part of the software in an entirely different area. So, a well-thought API is always a big advantage.

4.4.12.2 Metrics

Care must be taken that only the parts of the software that apply to the application at hand are activated and nothing else. Needs proof by analysis.

4.5 Software architecture characteristics

Again, generalising the following aspects of embedded systems is quite difficult. The way these characteristics are usually verified is through inspection of design, code or test cases.

4.5.1 Software Architecture

4.5.1.1 Common characteristics

Embedded software is usually organised in several layers. Practice has shown that in many cases, the number of layers turns out to be three:

- I/O hardware access, operating system;
- Middleware, not really tightly linked to hardware, but providing basic services such as device drivers and interrupt handlers. Access to this layer is done through an API. A very interesting approach is to implement a "virtual machine" at this level. This way a nice path towards portability is laid down;
- Top layer contains the functionality of the application.

The main advantage of such an organisation is that general and application-dependent features can be kept apart quite nicely. Also, if needed, different software development tools can be deployed per layer.

4.5.2 Scheduling Type

4.5.2.1 Common characteristics

This is very dependent on the type of constraints that were put forward: hard or soft real-time requirements. For hard requirements, a time-triggered approach is to be preferred, because it is a lot easier to prove that the software has a predictable behaviour. When reaction time to external influences is of the highest priority, and hard requirements are not present, then an event-driven architecture might give better results. A mixed approach is also possible.

For some applications, many events have to be handled in parallel, so yet another scheduling is required: concurrency. This can be achieved in close interaction with an operating system that has this functionality, or a static or dynamic scheduling scheme can be compiled directly in the application. It will become apparent from what follows that this choice will have many consequences.

4.5.3 Interrupt handling

4.5.3.1 Common characteristics

When hard real-time requirements have the highest priority, then the frequent use of interrupts should be avoided as much as possible, because they have a negative influence on the predictability of the software. A polling-scheme or other time-triggered approach is better.

4.5.4 Tasking and Exception Handling

4.5.4.1 Common characteristics

The same remark holds for these features, although an exception would be allowed to signal hardware failures.

4.5.5 Processor privilege usage

4.5.5.1 Common characteristics

In most cases, the mode with most privileges is used to allow direct and strict control over all hardware, which again is improving the predictability. The use of dual modes can be beneficial when strict software partitioning is required.

4.5.6 Languages used

4.5.6.1 Common characteristics

As was to be expected, a large variety of languages are used: from assembly language over C to object-oriented languages such as C++, Java and Ada. The code is often implemented using a low-level language such as assembly, because of the need for performance optimization (e.g. timing and power consumption constraints) [40]. This makes the code hard to understand and monitor for quality and consistency [39], is more error-prone and needs low-level code implementation testing more than higher-level languages [41].

Also, modelling languages such as UML and SDL are in use, or ESTEREL for creating finite state machines. For designing GUIs, graphical rapid prototyping are used quite often.

It would seem that when safety-criticality aspects of the software have higher priority, there is a shift towards more abstraction: higher-level languages and modelling descriptions. Although the costs involved are usually higher, and learning curves are steeper, the long-term benefits are worth it.

4.5.7 Modularity

4.5.7.1 Common characteristics

By carefully separating the different functionality of the embedded software into different modules, the way lies open to define a consistent, recognisable template that can be reused throughout the whole application. By maintaining strict rules and discipline in the implementation of such an approach, reuse and a full-fledged component approach are within reach.

4.5.8 Portability

4.5.8.1 Common characteristics

The development environment and the language constructs used should ensure portability as much as possible, to avoid software changes each time the hardware is upgraded or replaced.

4.5.9 Configurability

4.5.9.1 Common characteristics

Whereas a previous section already discussed this feature to some extent, it only dealt with the static aspect of changing the configuration of the embedded software. This time, the dynamics of the program sometimes need to be changed on the fly, at runtime. If the changes control a part of the software that steers hardware, then the system may behave quite differently after such a change.

4.5.10 Operating Systems and Kernels used

4.5.10.1 Common characteristics

Again, the architecture choice impacts this aspect a lot. If requirements disallow the use of a number of features, then an appropriate choice of OS and/or kernel can make the embedded software application more deterministic, and can be a big help in proving that this is the case. Use of a stripped-down kernel, or even eliminating the need for a kernel can be a valid choice. New research efforts about component-oriented run-time layers proves very promising, but also the more "classical" real-time operating systems have their merit, depending on the type of application, because they provide extendibility towards the future.

Care must be taken when using the memory features of the processor or the operating system. For some application, it is considered no problem to make use of virtual memory and/or cache. Other applications can not allow those features to be used, again for reasons of predictability.

4.5.11 Hardware architecture characteristics

Due to the large variety of possible used hardware components and architectures, generalisation and measurement is quite difficult, if not impossible and was therefore omitted. The characteristics of the hardware usually follow from:

- The specifications of the customer
- Decisions made in the design to meet the specifications
- Conclusions deduced from a prototype.

4.5.12 Development tools

4.5.12.1 Common characteristics

In embedded software development, the use of custom hardware and real-time and performance constraints result to a limited set of development tools which can be used [40], and for example, can force the project to develop their own tools if suitable ones are not available [41]. In addition, most development environments and tools for creating and debugging embedded software are primitive compared to equivalent tools for richer platforms [42, 43]. Various commercial and also experimental environments are used to control several aspects:

- High-level design tools + code generators
- Language-specific development environments
- Graphical rapid prototyping + code generators
- Configuration management tools
- Test case generation packages
- Documentation generation packages
- Scheduling tools
- Problem reporting/tracking tools
- Requirements tracking tools.

4.6 Embedded Agile Software Development

4.6.1 Unifying Embedded and Agile Software Development

Recently a number of new challenging but very promising research directions were opened, unifying agile software development methodologies with analysis and modelling, with more rigorous plan-based methodologies such as UP and MDA, with standards and certification, such as CMMI and DOD-STD-2167, and with real-time and embedded system development. Since most agile methodologies focus on code and neglect the higher-level software development activities such as analysis and modelling, a crucial part of the software development process is rather neglected. Recently, a number of attempts have been made to integrate analysis and modelling with agile principles, resulting in e.g.. Agile Modelling [2] and extreme modelling [44].

Although the Unified Process (UP) and agile software development seem to be quite contradictory on first sight, they are not necessarily exclusive since synergisms are possible and can offer the best of both worlds [45-47]. By incorporating techniques from both methods, an innovative process seems possible that delivers better quality software quicker than today. Since the Unified Process is a process framework, it can be tailored to the project discriminants that agile methodologies explicitly established and that are compatible with the overall UP superstructure. As such, important agile practices can be incorporated in the development process, while still acknowledging the importance of architecture, model abstraction and risk assessment. An Agile-oriented RUP instantiation can thus be made as light- or heavy-weighted as desired (in artefacts, deliverables, formality, rules, processes, …), as even Grady Booch himself explains in describing dX - A minimal RUP Process

[48]. Regarding MDA, current research is trying to apply and integrate agile approaches within the MDA framework, resulting in a Agile Model-Driven Development (AMMD) approach, a more realistic and light-weight down-to-earth approach as an alternative to very complex MDA-related tools and methodologies [49-51].

Regarding standardization, current research is focusing on integrating agile methods within a CMMI-standardized environment [52]. While standards provide guidelines what to do at an organizational level, agile methods can provide solutions to how to do it, how to develop software at a project level. Together, these models could form an innovative, comprehensive framework for structuring the software development organization. 0n the other hand, standards are also evolving and adapted in order to enable IID (iterative and incremental development) [53]. For instance, the DOD-STD-2167 standard was updated to DOD-STD-2167A and later to MIL-STD-498 in order to be able to perform evolutionary development.

Since embedded software development has a number of specific requirements and characteristics, embedded software development differs significantly from classical software development. This means that the agile development methods and practices as applied for classical software development can not identically be copied for embedded real-time software development, but have to be evaluated, adapted, extended and integrated with proven embedded software development techniques in order to meet its specific requirements. Although embedded systems are quite an unexplored domain field for the application of agile methods, recently research is being done to investigate how and which agile practices can be applied within embedded real-time software development [35] [54].

5. EMPIRICAL BODY OF EVIDENCE: STATE-OF-THE-ART

Authors: Hanna Hulkko, VTT
Tuomas Ihme, VTT

5.1 Introduction

This chapter contains a summary and review of the existing experiences on adopting agile methodologies or practices in embedded software development projects. The document is based on a literature survey, in which the existing experiences have been collected.

A brief overview on the challenges facing embedded software development is given together with motivation why agile methods should be adopted in the field of embedded systems development. Then, the existing experiences of adopting agile practices in embedded systems development projects are presented according to different stages of software development lifecycle. Motivation why to use a specific agile practice in embedded software development is provided together with the challenges of its adoption resulting from the characteristics of the embedded domain. Additionally, positive and negative experiences from the practice's adoption are summarized together with practical advice on how to apply the practice, and how the practice can be modified to suite the needs of embedded software development. Then, the reported experiences of all practices are summarized, and shortcomings of the current knowledge are discussed together with identified needs for future research. Finally, the identified future research needs are mapped with the scope and goals of the AGILE-ITEA project.

5.1.1 Challenges facing the development of embedded software

For a large embedded software project a hybrid method blending and balancing the features of different agile and non-agile methods and practices is often the choice due to varying characteristics of the different parts of the product [55]. Software methodologies should today provide a capability to develop, deploy, and evolve complex real-time and embedded systems whose requirements, platforms and operating environments are only partially fixable and may change unpredictably. The following special challenges will be discussed in more detailed: changing hardware requirements, unavailability of target hardware and presence of large, heterogeneous teams.

5.1.1.1 Changing hardware requirements

In embedded systems, a change in one component (hardware or software) usually causes a change in another [56]. Often, the hardware requirements and design are incomplete at the beginning of the project, which leads to requirement changes on software created by hardware [35, 39]. The changes in the hardware design and interfaces are often adapted with software, since changing the software is cheaper and easier [57]. After the hardware is ready, fixing and tuning the system are made with software as the needs are discovered, which again means unexpected requests and changes [57]. These changing requirements often cause also delays in project schedule, which can cause bottlenecks in embedded software projects, where e.g. the hardware development is incomplete when the software is ready for testing [56]. All these issues together with high costs and practical limitations of after-release corrections require high effectiveness of pre-release change identification [39].

5.1.1.2 Unavailability of target hardware

In embedded systems development project, the software and hardware are usually developed concurrently, and thus, the target hardware is not available until late in the project [33]. This causes problems for testing the software, e.g. by making early integration testing difficult [39], and creating a need for using simulators for testing purposes.

5.1.1.3 Presence of large, heterogeneous teams

The software developers in embedded systems development projects have often background in electrical engineering [57] unlike traditional software engineers. In addition to the software teams involved in embedded systems development, hardware, mechanical, and business teams are often present. Since the embedded systems development projects can require tens or hundreds of person months and input from numerous areas of expertise, the teams are often large. Additionally, the teams can often be geographically distributed between different sites [35]. In addition to the presence of heterogeneous teams, the concurrency and sharing of software parts places high demands on communication between the different technology groups. Thus, shared communication mechanisms, information distribution and distributed decision making are very important in embedded system development [39].

5.1.2 Motivation for using agile methods in embedded software development

Some of the characteristics of embedded systems and software development discussed in Chapter 4 and also in the previous section create a special need for the adoption of agile methods and practices. As one of the main principles and motivation behind all agile methods is responding to changes in e.g. requirements or technology [58], the volatile nature of embedded software development, which is mainly caused by the changing hardware requirements, can be obviously accommodated with the use of agile methods [35, 57, 59]. The iterative development approach and periodical re-evaluation and planning of the project and its goals enables the embedded software developers to respond to the changing hardware design. In addition, it decreases the probability of after-release changes, which is important since most embedded software programs are stored in ROM, and thus it is very expensive to change the software once the final image has been transferred onto the chip [56]. The iterative and incremental design approach is also useful in the development of safety-critical software, since it enables periodical re-evaluation of safety-issues and helps to find failure scenarios [60].

Another characteristic of embedded systems development in favor of adopting agile methods is the elevated importance of testing [35], as testing is also a central activity in several agile methods, such as eXtreme Programming [3], Lean Software Development [8] and Dynamic Systems Development [10]. According to Grenning [33], unit and acceptance testing practices of eXtreme Programming enable embedded software developers to make meaningful progress prior to hardware availability. Removing defects from the product through resilient testing in different lifecycle stages also increases the reliability of the end product, which is very important in many embedded systems (see Chapter 4 and also the previous section). Additionally, testing helps to minimize the probability of after-release defect corrections, which are very expensive and difficult because they are often mass-produced [40]. However, the embedded system development places also specific constraints to testing (see section 5.5), which can affect the feasibility of the agile testing practices.

Third aspect of agile methods, which makes them well suited for the needs of embedded systems development, is their emphasis on communication and collaboration. Since embedded systems development involves large, multidisciplinary and often geographically distributed teams (see the previous section), there is a need for efficient communication between the different teams, especially hardware and software developers [35]. Also, embedded software development projects can have several stakeholder groups, such as partner developer company, internal management, internal and external customers, mechanical and hardware engineers, and adopting agile practices such as the Planning Game can simplify the software team’s interface with the stakeholders [61].

Finally, the hardware engineering backgrounds common to embedded software developers can be a source of change resistant attitudes and on the other hand, a reason for lack of knowledge on general software development activities and practices. According to Greene [57], the attitude problems can be overcome by adopting agile practices, since the agile principle of barely sufficient process decreases the reluctance to adopt new practices and tools, and the agile principle of valuing people over processes helps to overcome the developers' aversion to oppressive processes. This is also supported by Morsicato and Poppendieck [60] who argue that XP practices are adopted more voluntarily than "traditional" software development processes, because XP contributes and improves the developers work directly, and does not feel like an extra burden forced by the process improvement instances. Additionally, adopting agile practices can be an effective way to transfer some of the best practices of software development to the developers with hardware engineering backgrounds [57], and thus increase their knowledge on software engineering.

5.1.3 The experiences of adopting agile practices in embedded software development

The experiences of adopting agile practices in embedded software development projects are presented in the following sections according to subsequent lifecycle stages of software development, i.e. planning and project management, design, implementation and testing. The agile practices, of which there are reported adoption experiences, are discussed one at a time under appropriate lifecycle phases. The following issues are covered:

- **Motivation**, i.e. to which special characteristics or needs of embedded systems development the practice can contribute to
- **Challenges**, i.e. special characteristics of embedded systems development, which might impede the adoption of the practice
- **Positive experiences**, i.e. reported benefits gained from adopting the practice
- **Negative experiences**, i.e. reported drawbacks faced when adopting the practice
- **Suggestions for adoption and use**, i.e. practical advice on how to apply the practice, and how the practice can be modified to suite the needs of embedded software development. Also, additional practices ("embedded agile" practices) suggested to be used in embedded software development projects to support the adoption of agile practices are presented here, if they have been reported.

A summary of the experiment reports, from which the positive and negative experiences can be found in Appendix 1. Additionally, the summaries of Appendix 1 have been utilized in Section 5.6, where the types of development projects where agile practices have been adopted are discussed, including e.g. different application domains and product types.

5.2 Project management and planning

5.2.1 Planning Game (iteration planning)

Planning Game is a joint session between the customer and developers where the contents of the next iteration is planned. In XP, Planning Game is held in the beginning of each iteration. [3]

Practice	**Planning Game (iteration planning)**
Motivation	- Embedded systems development is unpredictable due to potential need for performance optimization and effort needed to find complex, hidden bugs [40]. Thus, iterative planning is more feasible than trying to predict the whole project in the beginning. - The final split between HW and SW functionality is done at a late stage

	of the development, and the software may have to be tuned to meet technological constraints related to memory and power consumption. This is why SW requirements cannot be frozen initially, but they should be refined iteratively. [35] - The incremental planning enables the periodical re-evaluation of safety issues and helps to find more failure scenarios, which is needed in safety-critical systems development [60]
Challenges	- Majority of embedded systems software is not directly visible to the real customer, which limits the possibilities and value of communication between the developers and end customer [40]. There's often only very few user stories but each user story comprises a multiplicity of complex functions. - Since embedded software can have limited interaction with user, development is often based on architecture and high-level design specifications rather than user requirements [40].
Positive experiences	- Planning game simplified the team's interface with multiple stakeholders (partner company, internal management, internal and external customers, mechanical and HW engineers) [61] - Including team members to making estimations increased commitment [62] - Direct communication with the customer improved visibility and improved the team's attitude. Also, a clearer definition of expectations was obtained. [62] - Helps to avoid unattractive, wrong or obsolete features [55].
Negative experiences	- End user not interested from software or a part of software, but from the whole embedded product, and thus getting priorities from the end customer is not feasible [63] - Only 80-90% of the requirements were captured at planning game, rest were discovered during implementation (due to lack of end customer) [64]
Suggestions for adoption and use	- A marketing team with technical expertise or engineers with business expertise can act as a proxy for the real customer to define and prioritize the features for the project and next iteration, because the actual customer might not understand or be interested in the technical decisions, because they are not necessarily visible directly in the end product. The real customer and the proxy should frequently discuss the overall development to make sure they have the same visions for the system. [40] - For the initial development cycle (early iterations), do an up-front product feasibility study. For follow-on development cycles, do an up-front feasibility study. Establish general long term plans focusing on the decomposition of the project into small components and stories. [40] - Hardware team can be used as customer to define and prioritize requirements with [63] - HW requirements should be fixed first, and then SW requirements can be defined and planned as in XP [65] - Since the most critical requirements should be implemented first according to XP, this means planning and implementing the device drivers first (real-time requirements depend on device drivers, HW needs to be verified for production with drivers, and testing needs I/O functionalities) [65] - Stable and volatile parts of SW were identified, and long term planning was made to the stable ones, but the volatile parts were planned only one iteration ahead [62] - If dedicated customer is not an option, majority of requirements should be established before the first iteration [64]

5.2.2 Short Iterations / Small releases

In XP, new releases of the system are made in short cycles, which span from days to couple of months [3]. The same principle of short iterations is also a central part of the whole agile philosophy [58]. Additionally, the small releases support the concept of small and incremental changes important in agile software development.

Practice	**Short Iterations / Small releases**
Motivation	- The software requirements change constantly due to hardware changes in an embedded project. Short iterations provide a good possibility for adjusting to these changes. [35] - Embedded software development is experimental in the beginning, but the practices should turn more rigorous as the project progresses due to the increasing area of impact of SW changes [35]. Short development cycles provide a good possibility for adjusting the practices.
Challenges	- Decomposing the functionality into small releases can be hard, because embedded systems are often monolithic systems (i.e. do not function without all components), which are composed of several, distinct-purposed components. [40]
Positive experiences	- Smaller releases enabled early risk alleviation by providing a possibility to test tools while producing a limited set of functionality in the first iterations [64].
Negative experiences	- Because development was focused at one feature at a time, feature's interaction was not considered enough and thus the most complex tasks were faced at the end of the project [64].
Suggestions for adoption and use	- Release schedule was integrated to system integration and testing schedules [64]. - 3-month release cycle chosen so that one feature could be done in one iteration [64]. - Extra-long cycles may be required at the start of the project. A large amount of work is needed up-front to do detailed feature decomposition plans to allow for rapid cycles. [40].

5.3 Design

5.3.1 Simple Design

According to XP, the software should be designed in the simplest way, which meets the requirements [3]. Aiming to simplicity is also a part of the overall agile philosophy [58].

Practice	**Simple Design**
Motivation	- There are inherent needs for simplicity in embedded software design: - Although embedded systems are layered (HW, HW-specific SW and application SW), over-design of layers should not be done [65] - The need for an operating system should be assessed for cost-benefit, and the use of operating system avoided for simplicity if not necessarily needed [65] - Software design should be started even if no HW exists yet by defining the SW/HW interfaces in the simplest possible way, and developing the SW to work with the interface description [33]
Challenges	- Up-front design is particularly important in embedded systems development for meeting specific requirements such as real-time

	performance and portability [40] - Some design decisions which have to be made early in the development, like HW choices and division of work among components, should be done based on careful design, because they are very hard to change later [40] - Preliminary architecture design has to be done, and design is driven by performance issues rather than adding new functionality [35] - Distributed development and heterogeneous teams are reality in embedded systems development: up-front design documentation is needed for effective communication and effort synchronization [35] - In large, complex systems a predefined architecture is needed, and it should be as stable as possible. Thus, refactoring is not enough, but a design needed. Also, models should be utilized to enable automatic code generation. [66]
Positive experiences	- XP helps to focus on the simplest solutions that are necessary now [61].
Negative experiences	- If more time had been spent on design, high-risk areas of the code and interaction of the features might have been identified better (in a mission-critical software development project) [64] - Non-critical assembly code should not be optimized, because it causes poor maintainability [57]
Suggestions for adoption and use	- For the initial development cycle (early iterations), senior architects create an up-front initial system architecture. For follow-on development cycles, senior designers involve in assessing whether the previous cycle's architecture is reasonable. [40] - Keep a balance between extensible components for future features and designing and coding the current feature [40]. -Up-front designing [61], [67]: -- Initial, non-agile architecture was not very detailed and remained useful in the XP process -- In the beginning of each iteration, the design was revised due to the features that were added -- Designing for the current iteration and making some provisions for the one after that -- Detailed designing in order to have valid estimates -- Designs were analyzed using several use case scenarios. - HW-related issues were planned as far as possible [61] - In mission-critical environment, requirements rarely change because they have been diligently defined in a contract between the customer and business team prior to the project's inception and thus, 80-90% of all requirements should be completely defined prior to the first iteration [64] - Device drivers should be written to work in current product, and not try to make generalizable code (but places for generalization can be identified in comments) [65]

5.3.2 Light Documentation

Placing emphasis on working software over documentation and minimizing the amount of unnecessary work is a principle of all agile software development methods [58]. For example, in Lean Software Development, "Eliminate waste" is one of the seven principles [8].

Practice	**Light Documentation**
Motivation	-

Challenges	- In embedded systems development, distributed development and heterogeneous teams are common: up-front design documentation is needed for effective communication and effort synchronization [35, 66] - Relying on self-documenting code through code commenting can be unfeasible due to performance tradeoffs caused by the comments [40] - Long lifetime of products places requirements for explicit documentation (also because possibilities for self-documentation are limited)[40] - Lack of architectural and other documentation can limit the possibilities for refactoring [40] - Certain embedded systems are subject to regulatory requirements regarding documentation [40] - In the development of safety-critical systems, emphasis is placed requirements traceability [60], and thus extra documentation is required
Positive experiences	- XP prompted to ask the following questions about the software design document at the start of each iteration: “Is this necessary now” [61] - Focusing more on coding enables early cycle and memory usage metrics to be taken from the initial code in a development stage, where HW choices can still be changed if necessary. The metrics also facilitate project course corrections and more accurate planning. [40]
Negative experiences	- The Light Documentation practice relies much on tacit knowledge which may cause problems if the project team changes [55].
Suggestions for adoption and use	- Create artefacts for internal use documenting how to use software and hardware [40]. - Produce documentation based on the introduction of the initial junior staff to the project and use this documentation in introducing new junior staff [40]. - Brief developer-level documents, “Mini-docs”, were created by the project team as needed for promoting knowledge transfer around the team. [67]

5.4 Implementation

5.4.1 Continuous Integration

In XP, the code is integrated and tested after few hours to a day of development. [3]

Practice	**Continuous Integration**
Motivation	- Long integration periods can make the integration very complex and time-taking due to numerous separate code bases of different product variations [63]
Challenges	- Embedded software development environments may lack appropriate tools needed for the integration, e.g. integration can take too much time [63] - In safety-critical systems development, integrations may need to be controlled to assess the effects of the changes, and spontaneous continuous integration may not be feasible [64]
Positive experiences	-
Negative experiences	- The used tool built new code base slowly and locked the code until the integration was ready, so developers did not integrate frequently [63]
Suggestions for adoption and use	- Developers should do code integration to the code base immediately after unit testing, and integration test should be ran at nights [63] - In one safety-critical systems development project, all integrations were

	be assessed and accepted by a separate change control board who also assured that the change requests had been fulfilled. This board met bi-weekly, which caused a longer time period between integrations. These longer integration periods enabled to think about the effects of a change in detail, but on the other hand, made the integration harder and more time taking. [64]

5.4.2 Pair Programming

XP suggests, that at least all production code is written by two people using one computer. [3]

Practice	**Pair Programming**
Motivation	- Enhances communication, which is especially important between HW and SW developers in embedded projects [35] - Helps to disseminate knowledge on different areas of the system to the developers, which is important in embedded projects because of several different technology domains
Challenges	- Pair programming can be perceived as a radical change and met with resistance, which is especially the case with embedded SW engineers, who are more resistant to process changes [40] - The developers of embedded systems can work in shifts to minimize the effects of HW shortage [63], which can complicate finding common time for pair work
Positive experiences	- Pair programming found very beneficial, as it was accompanied with code reviews, which became more valuable and active, as everyone new the code they were reviewing [61]
Negative experiences	- Cross-training benefits were not as extensive as hoped, because the need for specialized domain knowledge in embedded system development too vast for anyone to have [57] - In complex environments, experts and feature owners are needed over common knowledge (knowledge transfer benefits of PP not as extensive or even important) [64] - High initial adoption, which decreased during the project, and finally, only risky changes were made in pairs. Formal reviews were replaced by pair programming at first, but then brought back due to low usage of PP, missed defects and organizational pressure [64]
Suggestions for adoption and use	- In assembly coding, pair programming has value only in the initial development stages (detailed design and initial coding) [57] - Pairs worked on design and complex coding tasks, simple and repetitive tasks were done solo in a co-located office space [63] - Co-ordination was needed to match pair member's working times [63] - In mission-critical environment, pair programming should be accompanied by formal code and test case reviews [64] - Cross-Functional Pair Programming is a modification of pair programming, in which software and hardware developers are paired to create all or a part of an embedded system. It is most beneficial when the embedded system includes new or unused HW, and both pair members have knowledge of each others areas. [68]

5.4.3 Refactoring

Refactoring means improving the structure of code without changing its functionality, and is one of the practices of XP [3].

Practice	**Refactoring**
Motivation	- Useful in performance optimization which is important in many embedded software design due to resource and timing constraints [40]
Challenges	- Cannot be done in the same way at the end of a HW design cycle than at SW cycle, because the HW gets gradually more fixed [69] - Affects timing, and can thus be hazardous in systems with real-time constraints [35] - Limited by specialized areas, lack of architectural and other documentation, and partitioning of the functionality to separate units (e.g. processors) [40]
Positive experiences	- Refactoring helped to find failure scenarios in safety-critical software development (also root-cause analysis of found bugs proved, that when team felt something should have been refactored, they were right) [60] - Refactoring reduced the amount of “kludginess” in the code and improved the quality of legacy code [57]
Negative experiences	- Before XP, root cause of most bugs was code reviews, and after, insufficient refactoring [61] - Large refactorings created significant defects (too much reliance on test suite to detect possible defects caused by refactoring) [64]
Suggestions for adoption and use	- Performance optimizations should always be done based on measured problems, not speculation [33] - In safety-critical software development, where the effects of refactoring have to be carefully assessed, the developers wrote down identified refactoring needs to a “wish list”, and the permissions to refactoring were granted by a separate change control board [64]

5.5 Testing

5.5.1 Unit testing

Practice	**Unit testing / TDD / Test first**
Motivation	- Embedded systems often have high requirements for reliability, exception handling correctness and mean-time-to-failure, and thus, systematic and rigorous development processes and testing are needed [40] - In safety-critical systems development, XP’s testing practices help to assure, that the safety controls are not violated when the system is changed [60] - Assembly coding used in many embedded systems is more error-prone and needs low-level code implementation testing more [41]
Challenges	- Many embedded systems do not have the devices (e.g. display) or memory (to store the test code) required for unit testing [65] - Daily testing not necessarily possible due to shared HW simulators (with HW teams) [35] - Memory and performance issues can prevent running all test code at the same time in the test environment [35] - Use of test code can be limited by memory constraints, or it can alter timing and hide errors [40] - Use of breakpoints or single-stepping can be blocked by timing constraints and multi-tasking [40] - HW simulators are expensive, take time to implement and can be slow; dual targeting to both simulator and real HW takes extra effort [40]

	- Regression tests require test harnesses and equipment, and it is difficult to verify the internal state of the system for correctness [40] - Usually in embedded software development, no OO languages are used. Thus, lack of tools for unit testing can make them hard to automate, so emphasis should be put on acceptance tests. [33] - Incompatible compilers of development and target environments slow development down, because the code has to be downloaded to the target system after each compile, and this can take time. This makes it also harder to port a unit test tool. [33]
Positive experiences	- Resulted in increased quality, ease of coding, more focused planning and design, and enabled timely feedback [33] - Unit tests were the most useful of the 13 adopted agile practices, because they enabled testing components at a low level, which was needed especially as assembly was used as the programming language [57] - In large mission-critical system development, 1500 unit tests written and merged into a test suite, which was perceived very useful and critical to refactoring [64] - TDD resulted in lower bug rate [59] - TDD was easy to adopt [61]
Negative experiences	- If simulators are used for unit testing, the real peripheral HW cannot be used in testing [65] - Manual checking of the code enabled easy separation of typos from actual bugs [70] - Automated testing requires more accurate syntax use and more detailed test cases than manual testing [70] - Unit testing was not mandatory, so many developers did not use it [63] - Existing tests were not updated according to code changes, so there was nothing to test refactorings against [63] - Large test suite takes a lot of effort to maintain according to changes, and thus, sometimes the design decisions were affected by unwillingness to make major changes to the test suite [64]
Suggestions for adoption and use	- Unit testing can be done by running test scripts on PC, which has been connected to the HW with a serial port. Scripts can be made, which send messages through the serial line and collect the responses, and then the responses can be compared to expected ones. [65] - In real-time systems development, unit tests can be written to check execution times, but usually HW is needed for actual testing [33] - Due to the use of different development and target environments in embedded software development, dual-targeting (i.e. enabling the SW to be compiled and ran in both target HW and development PC) should be done to enable early (unit) testing and simulations. [33] - Trouble log which used little memory and was fast to execute was used to obtain error descriptions [59] - Dual targeting (i.e. ability to run the SW on both target HW and PC) was used to separate SW and HW bugs and to enable automation of unit tests [59] - HW-related unit tests were used to test HW controllers and drivers independently during development, and by HW vendor as final test suite before shipping [59] - In a project where part of a legacy system was implemented, a test suite for testing interfaces with legacy code would have been necessary in addition to the “unit test suite” [64]

5.5.2 Acceptance testing

Practice	**Acceptance testing**
Motivation	- In real-time systems development, acceptance tests can be written to simulate loads and event sequences, and then performance (e.g. CPU usage) can be measured [33] - System bugs are not automatically blamed on software, because it is easy to test whether the software actually causes them or not. This also improves the communication between SW and HW engineers [59]
Challenges	- Memory and performance issues can prevent running all test code at the same time in the test environment [35]
Positive experiences	
Negative experiences	- In a mission-critical project, there was too much reliance on the test suite. Instead, a balance between code reviews, automated regression tests and acceptance tests should be used [64]
Suggestions for adoption and use	- Acceptance tests must be automated as much as possible by replacing outer SW layers with a test interface. As much as possible should be tested without HW and manual intervention. [33] - Domain tests were used to test separate domains, when unit testing was too fine-grained and system testing too high-level. One domain was built and loaded to HW, and I/O from rest of the system replaced by mock objects. This was used e.g. for tracing timing problems [59]

5.6 Summary of experiences from different agile practices

In the previous sections of this chapter, existing experiences on the adoption of agile practices in industrial embedded systems, and mission- and safety-critical systems development projects were presented together with motivation and challenges for their adoption. Experiences from nine agile practices were discussed under five software development lifecycle stages. It is evident, that these nine practices represent only a part of all practices of numerous agile methods. Also, since the experiences on their adoption stem from less than ten projects, the coverage of different application domains, product types, team compositions and other characterizing elements representing the wide spectrum of embedded systems development projects, the coverage of different project types is quite limited. In rest of this section, the findings of the presented state-of-the-art review are summarized. First, the practices, of which there are reported experiences, are summarized, and their overall reported usefulness discussed. Then, the project types and application domains where the agile practices have been adopted are summarized (see Appendix 1).

As discussed earlier, experiences form nine agile practices from five lifecycle stages of software development were found from literature. Adoption of also other agile practices (such as Coding standards of XP and Sprint reviews of Scrum) were mentioned, for example, coding standards in ([57], [61], [63] and [40] and reviews in [40] and [64]. However, only little actual experience from them was reported. Table 2 summarizes the amount of experiences from the nine agile practices together with their reported main benefits and challenges, as discussed earlier in this chapter. The practices are arranged so, that the practice, of which there are most experiences is first in the table, and the one of which there are least experiences, is last.

Table 2. Summary of experiences from different agile practices

Practice	# exp.	Main benefit	Main challenge
Unit testing / Test first / TDD	6	Assembly coding used in many embedded systems is more error-prone and needs low-level code implementation testing more	Use of test code can be limited by memory constraints, or it can alter timing and hide errors

			Lack of tools, programming language and device restrictions
Pair programming	3	Enhances communication, which is especially important between HW and SW developers in embedded projects	Need for specialized domain knowledge in embedded system development is too vast, so cross-training benefits of pair programming are not as substantial
Refactoring	3	Useful in performance optimization which is important in many embedded software design due to resource and timing constraints	Refactoring affects timing, and can thus be hazardous in systems with real-time constraints
Simple design	3	Helps to focus on the simplest solutions that are necessary now.	Up-front architectural design is particularly important for meeting real-time performance and portability requirements and enabling partitioning of work
Planning game	2	Enables effective prioritization and selection of requirements based on the latest hardware design	Planning with customer may not be feasible, because the software part of the system is not visible/interesting to the customer
Continuous integration	2	Long integration periods can make the integration very complex and time-taking due to numerous separate code bases of different product variations	Integrations may need to be controlled to assess the effects of the changes, and spontaneous continuous integration may not be feasible (esp. in safety-critical systems)
Short iterations / small releases	1	Enable responding to changes in requirements caused by hardware changes	Breaking large functionalities into short releases might not be practical Synchronizing release schedule with hardware releases for system integration and testing can be difficult
Acceptance tests	1	Acceptance tests can be written to simulate loads and event sequences, and then performance (e.g. CPU usage) can be measured	-
Light documentation	1	Helps to avoid unnecessary documentation. Focusing more on coding enables early cycle and memory usage metrics to be taken from the initial code in a development stage, where HW choices can still be changed if necessary. The metrics also facilitate project course corrections and more accurate planning.	Explicit documentation is needed due to maintainability, traceability and communication issues

An important factor in the industrial significance of the results of this review is the information, on which types of projects the experiences have been derived from. Embedded systems development projects can be characterized using different attributes, such as their application domain, product type, team size, etc. Although in embedded systems development projects there are often many people involved, the size of the teams in the reported projects was only between 4 – 10 persons, except in one project, where it was temporarily increased up to 18 persons. Another common characteristic of embedded software development is the use of low-level programming languages as opposed to e.g. object-oriented languages. This was also the case in all of the projects included in this review: in every experience report, where the used programming language was given, it was either assembly, C, or a combination of both. Third way to characterize the projects where the experiences have been reported from is the application domain and type of the developed product. This information, collected from the experience reports summarized in Appendix 1, is presented in Table 1.

Table 1. Application domains of experience reports

Application domain / product type
Automotive industry: - Developing customer-specific functions to bus software - Developing the front end of built-in car multimedia system
Public safety communication systems: - Re-implementing a part of a mission-critical, soft real-time system
Pharmaceutical instrument: - Safety-critical system development
Processor firmware development (2 reports from the same project)
Real-time mobile spectrometer development (4 reports from the same project)
Embedded legacy product development (application domain not reported)

5.7 FUTURE RESEARCH NEEDS

5.7.1 Shortcomings of existing studies

In the collection and review of the experience reports on adoption of agile methods and practices in embedded systems development projects, some shortcomings were identified in the existing empirical knowledge. First, the amount of material related to using agile methods in the development of embedded systems is very small: in all, approximately twenty publications were found in the performed literature survey, of which less than ten were actual experience reports. In addition, the number of embedded projects where agile practices have been adopted is even lower than ten, since in two occasions, the same project had been reported in more than one report. Second, almost every reported experience is related to adopting eXtreme Programming, or a set of its practices, while other agile methods, such as Scrum and Feature Driven Development (see e.g. [11]), are left without attention. Third, majority of the reported experiences are anecdotal, i.e. no measured, quantitative data or metrics have been presented to describe the concrete effects of the adoption of the agile practices. Also, no scientific, empirical studies, such as experiments or case studies, have been made (or at least published) related to the subject. Finally, as discussed in Section 5.6, the experience reports cover only a limited amount of different application domains and product types from the field of embedded systems. Additionally, the teams of the reported projects have been quite small, and not distributed, so the applicability of agile methods to larger teams has not been addressed in the reports.

5.7.2 Future research needs

The literature survey and review of the existing experiences on using agile methods in the development of embedded systems showed that the current knowledge on the subject is at a quite rudimentary level. Thus, in future, systematic scientific research and additional industrial experiences are needed to obtain concrete evidence on the possibilities and limitations of using agile methods and practices in the development of embedded systems. The research should be performed on different application domains of embedded systems, which are important but yet unstudied, such as the field of telecommunications. Also, the effect of different types of business models and product types to the adoption of agile methods should be considered. Furthermore, the studies should not focus solely on XP and its practices, but include also other agile methods. The overall goal of the research should be determining the most suitable and beneficial agile practices and methods including "embedded" variants of them for embedded systems development and identifying possibilities for tailoring and integrating them pragmatically to suit e.g. certain domains, product types or development contexts.

In AGILE-ITEA project [71], this work will be done by devising an agile based assessment framework to be used in industrial projects to help identifying project-specific constraints and needs for adopting different agile practices. The assessment will be a part of an agile software development framework for the embedded systems domain including all relevant processes and tools. The framework will be customized for different application domains, and accompanied by a deployment model whose purpose is to facilitate the industrial adoption of the framework. Additionally, experiences from the adoption of the agile practices will be collected into an experience base for systematic and effective dissemination and utilization of the knowledge.

6. CURRENT STATUS ON AGILE METHDOS IN EUROPEAN SOFTWARE DEVELOPMENT ORGANIZATIONS: RESULTS FROM A QUESTIONNAIRE STUDY

Author: Outi Salo, VTT

6.1 Introduction

This document reports the results from a questionnaire study that was made during January-February 2005 for European software development organizations of Agile ITEA project. The purpose of the questionnaire was, for one, to collect the past experiences of industrial partners' of Agile ITEA project regarding the use of agile methods and practices in these organizations. This work is related to WP1: T1.2 in Agile FPP. Thus, the questionnaire was designed in order to systematically collect and analyze this information. In more general, however, this study aims at mapping how the European software development organizations currently use different Agile methods and practices and how useful and potential they are regarded in different project contexts.

The target group of this questionnaire was especially the project managers and software developers of industrial Agile ITEA organizations. The questionnaire was/is available on a www server at VTT Technical Research Centre of Finland[5]. The contact persons of 18 Agile ITEA partner organizations were requested to disseminate the www link of the questionnaire to the software developers and project managers of their software development projects– whether agile or non-agile ones. However, it was required that only one questionnaire would be filled for one specific project that was either a finished or an ongoing one. Also, multiple filled questionnaires were encouraged for each organization in order to gain different views in one organization.

6.2 Background of the Study

A total of 35 filled questionnaires were received from 13 out of the 18 target organizations from 8 European countries. Following Figure 10 illustrates how the number of filled questionnaires distributed between the respondent organizations. Thus, six of the target organizations filled out a questionnaire of one project, where as the largest number of responses from one organization was as high as seven. It should be noted, that the percentages are counted from the data points that were available on each topic. Thus, the missing data points do not skew the results.

[5] The questionnaire form can be found on http://cgi.vtt.fi/html/kyselyt/agile/

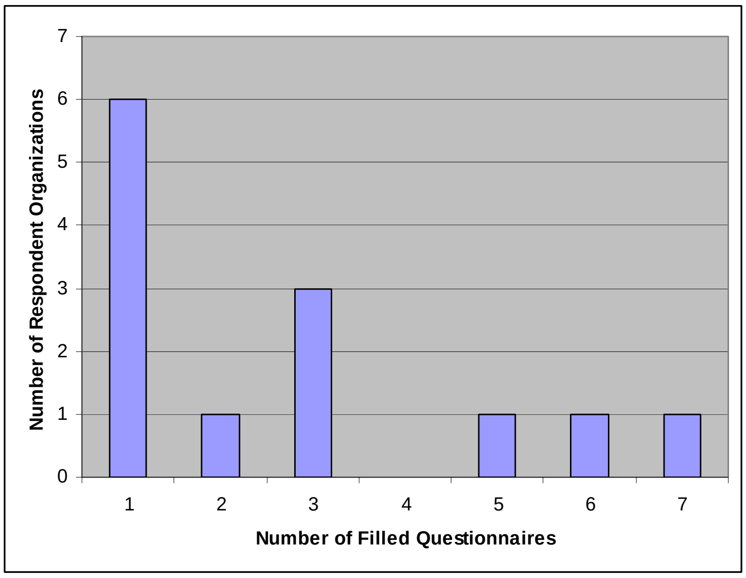


Figure 10. Frequency of Filled Questionnaires in Respondent Organizations

The focus group of the questionnaire was the personnel strictly involved in the daily software development activities, i.e. software developers and project managers. Following Figure 11 illustrates the distribution of respondents in following categories: 1) project managers, 2) software developers, 3) personnel that was responsible for managing the project but also participated in software development, 4) other, i.e. software process improvement and managerial personnel, and 5) missing information.

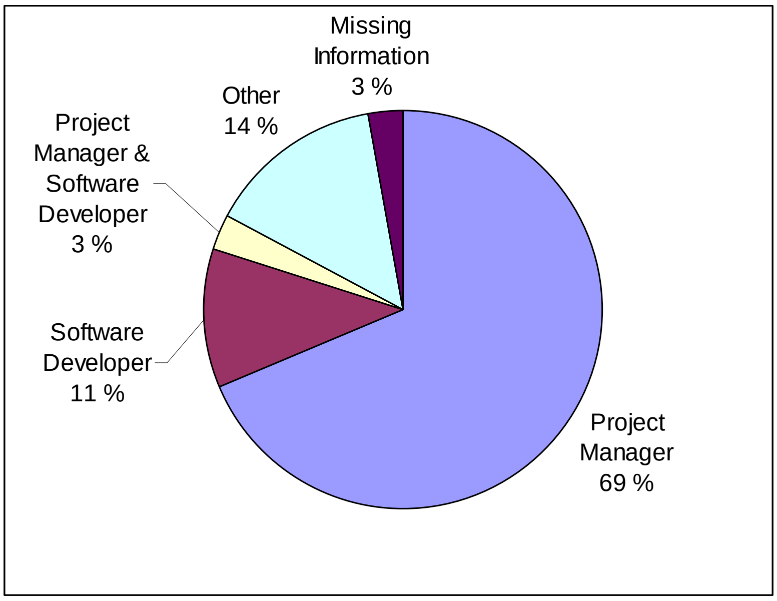


Figure 11. Role of Respondents in the Software Development Projects

The software development of the reported projects fell into three main categories: own vendor product development, development of a product for company internal use and component development for customer product. Figure 12 illustrates the division of the projects in these categories.

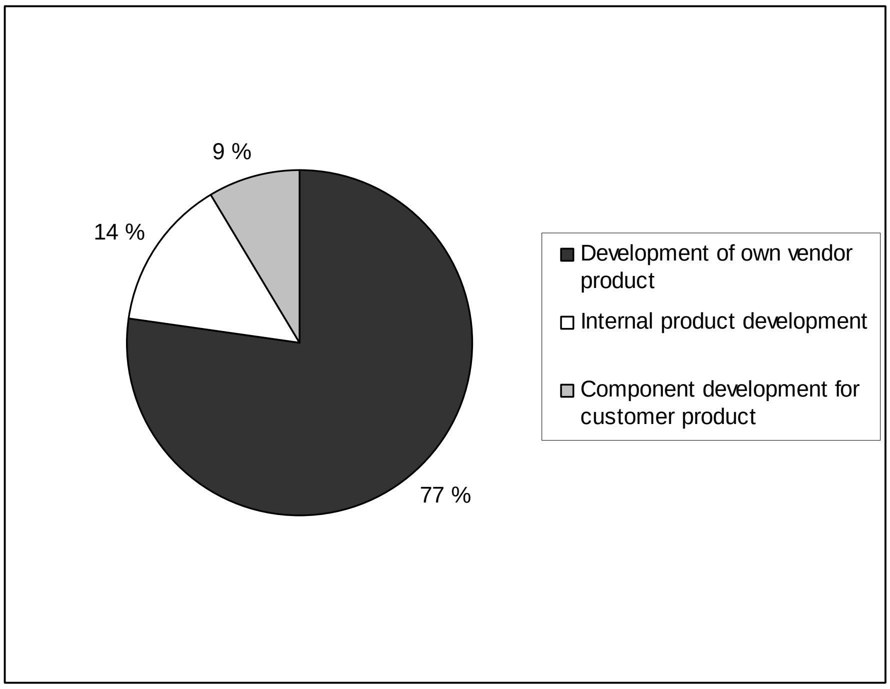


Figure 12. Frequency of Filled Questionnaires in Respondent Organizations

26 (i.e., 74.3 %) of the reported projects were currently ongoing whereas the number of finished projects was 9 (25.7 %). The duration - either estimated or realized - of these projects are illustrated in Table 3.

Table 3. Estimated/total duration of the projects

Project Duration	Frequency	Percent
2-4 weeks	1	2,9
between 1-2 months	2	5,7
between 2-6 months	8	22,9
>6 months	24	68,6

The majority of the reported projects (i.e., 68.6 %) lasted or were estimated to last six months or more. Accurately, the lengths of the projects reported in this category were reported to be:

1) over 6 months but under one year of duration (5 projects)
2) one year (3 projects)
3) over one year but under two years of duration (6 projects)
4) two years (5 projects)
5) three years (2 projects)
6) five years (3 projects)

The size of the reported software development projects is illustrated in Table 4. It can be seen that as much as 60% of the projects (i.e., 21) consisted of smallish (i.e., maximum of 10 developers) teams that are well suited in Agile software development context.

Table 4. Number of project members

Team Size	Frequency	Percent
<4	9	25,7
4-10	12	34,3
11-30	10	28,6
31-100	3	8,6
>300	1	2,9

One important aspect, regarding agile software development, is the criticality level of the application that is being developed. It should be taken into careful consideration when adopting and adapting agile practices for an organization. The criticality of the applications that are being developed in the reported projects are illustrated in Table 5. As it can be seen, the vast majority of the respondent projects develop a software product that, when failed, causes either discomfort for its user (32.4 %) or tolerable loss of money (38.2 %). Though failure of software product is always undesired, in 70.6 % of the projects the failure of its product, however, will not cause any permanent damage for its user. Originally, the agile methods (e.g., XP) are recommended and developed especially for this kind of software development [72]. However, these novel methods are developed consistently to include new ways of quality assurance. In the respondent projects, an effect of critical loss of money for the user was reported for 23.5 % of the projects where as the loss of a life or many lives was caused by only 5.8 % of the developed software applications.

Table 5. Criticality level of the application that is developed in the project

Criticality Level	Frequency	Percent	Valid Percent
discomfort	11	31,4	32,4
tolerable loss of money	13	37,1	38,2
critical loss of money	8	22,9	23,5
loss of life	1	2,9	2,9
loss of many lives	1	2,9	2,9
Total	34	97,1	100,0
Missing	1	2,9	
Total	35	100,0	

The managing of constant changes in, for example, product requirements during the software development process is one of the main benefits for adopting agile practices [73]. In Table 6 it is illustrated what was. As it can be seen, approximately half (51.5 %) of the reported findings fell into the category of "5-10%" and close to one third of the findings were in the category of "11-30%". Dynamism of the application requirements.

Table 6. Estimated requirements-change per month

Dynamism level	Frequency	Percent	Valid Percent
<5%	4	11,4	12,1
5-10%	17	48,6	51,5
11-30%	10	28,6	30,3
31-50%	2	5,7	6,1
Total	33	94,3	100,0
Missing	2	5,7	
Total	35	100,0	

6.3 Results of the Study

One of the main goals of the study was to enquire how the agile software development methods and practices are curretly used in a selected group of European software development organizations and how potential they are regarded as.

Thus, in the questionnaire the respondents were requested to evaluate

1) the current use of each of the listed agile related methods in the project, and
2) the usefullness of the method (in their opinion) (in the projects already using the method, the respondent was asked to consider how useful the method was in the project and, if the method had not been applied, to consider how useful it would be if employed in the project).

A five degree ordinal scale (Table 7) was used to evaluate the level of both the current use and usefulness aspects. The underlying assumption was that the respondents were somewhat familiar with the agile methods and practices. However, to avoid a situation of where respondent was not aware of the methods in question an option of “I do not know” was available. Also “not applicable” opinion was available.

Table 7. Scales for identifying the level of current use and usefulness of a specified method

Current use (the method/practice has been...)	**Usefulness** (the method is/would be... in my opinion)
1 = Systematically used throughout the project	1 = Extremely useful
2 = Mostly used throughout the project	2 = Very useful
3 = Sometimes used in the project	3 = Useful
4 = Rarely used during the project	4 = Not useful
5 = Never used during the project	5 = Harmful
6 = Not applicable	6 = Not applicable
7 = I do not know	7 = I do not know

In the sub-sections of this chapter, some of the most central and common methods, practices and issues are taken for further evaluation.

6.3.1 Agile Process Model

One of the central aspects and principles (http://www.agilemanifesto.org/principles.html) of agile software development is the iterative and incremental process model.

6.3.1.1 Iterative software development

Currently, 15.6 % of the respondent projects (5/32 projects) reported on systematically using short development iterations in their software development. In addition, 31.3 % (10/32) of the projects mostly used iterative development throughout the project. Thus, it can be said that close to one half (46.9 %) of the projects already applied iterative software development regularly. One fourth of the projects (8/32) reported applying software development iterations sometimes, 18.8 % (6/32) rarely and 6.3% (2/32) never. Only one of the projects (1/32) reported that iterative process model would not be applicable.

Table 8 illustrates the number of software development projects in crosstabulation of reported project and iteration lengths. The shortest projects (i.e., 2-4 weeks and 1-2 months) understandably consisted usually of only one development iteration. The iterativeness, however, is more visible in the longer projects, where only one project in categories of both “2-6 months” and “>6 months” reported only one iteration. For example, as much as 54.2 % (13/24) of the “>6 months” projects reported applying iterations maximum of two months that is in line with the suggestion of agile manifesto and its principles (http://www.agilemanifesto.org/principles.html).

Table 8. Crosstabulation of project and iteration lengths

Estimated/total duration of the project	Average iteration length of the project				
	<2 weeks	2-4 weeks	between 1-2 months	between 2-6 months	>6 months
2-4 weeks	0	1	0	0	0
between 1-2 months	0	2	0	0	0
between 2-6 months	1	2	4	1	0
>6 months	2	6	5	10	1

In 87.5 % of the questionnaires where the data was available (29/32) the usefulness of short software development iterations was regarded at least useful (i.e., 12.5. % useful, 46.9 % very useful, and 28.1 extremely useful). This reveals, that besides the projects that already regularly use incremental software development model (15/32), also a large number of the projects that currently only sometimes, rarely or never use iterative approach would regard it as useful to apply.

6.3.1.2 Incremental software development

Incremental software development aims at producing working software iteratively. In agile principles (http://www.agilemanifesto.org/principles.html) it is identified that “working software is the primary measure of progress”.

The respondents were asked three aspects on incremental software development:

1) current use of incremental software development,
2) usefulness of iterative development,
3) current use of “working software as a primary measure of progress”, and
4) usefulness of “working software as a primary measure of progress”.

The current use of incremental software development was reported as "systematically used" in one-fourth (25.8 %) of the 31/35 projects that provided data on this question. As much as 48.4 % of the projects (15/31) mostly used it throughout the project. Thus, the proportion of projects regularly working in incremental mode of software development was as high as 74.2 % (23/31 projects). 16.1 % (5/31) projects reported applying software development iterations sometimes, 9.7 % (3/31) rarely and 0 % never. None of the projects reported that iterative process model would not be applicable.

In 93.5 % of the questionnaires where the data on usefulness of incremental development was available (29/31) the incremental development was regarded atleast useful (i.e., 16.1 % useful, 35.5 % very useful, and 41.9 extremely useful). This, again reveals, that even the current level of applying incremental development mode is already high (74.2 %) in the respondent projects there is will among the project managers and software developers to use it increasingly in the future. Only 3.2 % (1/31) of the respondents regarded incremental development not useful and the same 3.2 % (1/31) as even harmul. This data was missing from 4 questionnaires.

"Working software as a primary measure of progress" ideology was systematically used in 34.4 % (11/32) of the projects and mostly used in 18.8% (6/32). Thus, over a half (53.1 %) of the projects of which this data was available (32/35) regularly use a working product increment as a measure of their progress. However, also one fourth (25.0 %) of the projects reported sometimes applying working software as a measure of progress. Only 15.7 % of the projects reported that they either rerely (6.3 %) or never (9.4 %) use such a measure. Also, one respondent (3.1 %) reported that this was not applicable in their project and one respondent (3.1 %) of not knowing the situation concerning this matter. This data was missing from three questionnaires.

In the questionnaires it was clearly visible that the respondents valued the ideology "working software as a primary measure of progress" and would like to apply it more in their projects. In numbers, this means that as much as 96.8 % of the respondents that answered this question (31/35) regarded this issue as, at least, useful (i.e., 19.4 % useful, 41.9 % very useful, and 35.5 % extremely useful). It should be noted that none of the respondents saw this issue as not useful or harmful. One of the respondents did not have opinion (i.e. "I do not know") on this issue.

6.3.2 Close communication

Close communication is emphasized in agile principles (http://www.agilemanifesto.org/principles.html):

> "Business people and developers must work together daily throughout the project."
>
> "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."

Extreme Programming (XP), for example, emphasized co-located software development teams and on-site customer. These two issues were enquired from the respondents of this study from the viewpoint of current usage and (potential) usefulness.

6.3.2.1 Co-Location of Project Teams

The location of the project teams was, at first, enquired in the project & application profile (in section 2. of the questionnaire (http://cgi.vtt.fi/html/kyselyt/agile/)). As much as 40.6 % (13/32[6]) of the project teams reported to work in the same open office space and 25.0 % (8/32) still in the same building. This, for one, supports the possibility of adopting agile software development methods, practices, and tools. 34.4 % (11) of the reported project teams were distributed either across the country or

[6] Please, note that in three of the questionnaires this data was not available.

globally. The current use of open-office space was also enquired in the practices and methods in organization (section 3 of the questionnaire in http://cgi.vtt.fi/html/kyselyt/agile/). This section provides slightly different viewpoint of this issue. It was found out that as much as 65.6 % of the projects use open-office space mostly (25.0 %) or systematically (40.6 %) in their software development. The open-office space was "sometimes used in the project" in 3.1 % (1/32) of the projects and rarely in 12.5 % (4/32). One of the respondents (3.1 %) found open-office space non-applicable in his project, and two (6.3 %) respondents did not have knowledge on this issue.

The usefulness of developing software in an open-office space highly regarded in the respondent projects. As much as 80.6 % of the projects which this data was available on (i.e., 31/35) regarded open-office space at least useful (25.8 % useful, 54.8 % very useful, and 16.1 extremely useful). One respondent regarded this kind of working environment not useful and one as eve harmful. Two of the respondents reported that open-office space would not be applicable in their project and two of the respondents did not have an opinion on this matter. Thus, when comparing the percentages of teams currently working in the same office space (40.6 %) and the projects that would regard it as useful (80.6 %) it can be said that there are, among the respondents, both willingness and possibilities to increase this kind of communication during software development.

6.3.2.2 On-Site Customer

Kent Beck suggests that in XP "a real customer must sit with the team, available to answer questions, resolve disputes, and set small-scale priorities " [72]. According to Beck, a real customer is someone who will really use the system when it is in production.

In "real life" of the respondent projects an on-site customer was systematically available of 9.1 % (i.e., three projects) of the 33/35 respondents. As much as 15.2 % of the projects (i.e., 5 cases) reported that an on-site customer was mostly available throughout the project. Thus, one fourth (24.2 %) of the respondent projects (i.e., 8) can argue on using an on-site customer. Using on-site customer "sometimes" or "rarely" in a project may be interpreted as using off-site customer: communicating with the customer that, however, is not present most of the time. This was the situation in 33.3 % of the projects (i.e., 21.2 % sometimes and 12.1 % rarely). On-site customer was not applied in 30.3 % of the projects and not applicable in 12.1 % of the cases. A total of 33 responses out of 35 were available on this topic.

Five respondents (15.2 %) regarded on-site customer as being extremely useful. One of these responses were based on systematical usage of on-site customer in the project. Two respondents had been using on-site customer mostly in the project, and one mostly. Also, one of the respondents who reported an on-site customer as "extremely useful" had not had one available at least on the current project. As much as 69.7 % of the responses were favourable on the usefulness of on-site customer (i.e., 30.3 % useful, 24.2 % very useful, and 15.2 % extremely useful). However, there were also four projects that reported on-site customer as not useful (12.1 %) and one as harmful (3.0 %). In the latter, the on-site customer was used "sometimes" during the project. Only one of the respondent projects which reported the on-site customer as "not useful" had been using on-site customer systematically where as the other four had never had one present. It is, however, impossible to evaluate if it is earlier bad experiences or some other issues causing the negative responses on on-site customer. Two respondents found on-site customer non applicable and three out of 33 available responses (9.1%) did not have knowledge on this issue.

6.3.3 Agile software engineering practices

6.3.3.1 Pair Programming

A controversial practice of XP is pair-programming [72]. Its effects on software quality [74] and productivity [75] has been widely studied. In this study it was found out that the regular usage of pair-programming practice is currently fairly low in industrial organizations (i.e., 3.0 % systematically, and

12.1 % mostly used throughout the project). As much as 27.3 % of the projects reported sometimes using pair-programming, 18.2 % rarely, and 33.3 % never. One respondent (3.0 %) regarded pair-programming as non-applicable and one as not knowing the situation of this matter. The respondent rate for this question was 33/35.

Very interestingly, also pair programming was credited as very potential practice in the future projects. As much as 75.0 % of the respondents (32/35) regarded that, if used, this practice would be at least useful (i.e., 40.6 % useful, 28.1 % very useful, and 6.3 % as extremely useful). As reported earlier, the projects of this study reported as much as 40.6 % (13/32[7]) of the project teams working in the same open office space and 25.0 % (8/32) still in the same building that supports adopting this kind of practice in their software development.

Only one of the respondents reported that pair programming would have been not applicable and three respondents (9.4 %) were unsure of its usefulness (i.e., "I don't know"). As much as 9.4 % of the respondents (3/32) regarded pair programming as not useful and 3.1 % as harmful (1/32). The situation of pair programming being harmful was being reported by a project manager on a project where pair-programming had been used mostly throughout a project. In all the projects where pair programming was reported as "not useful" it had not been applied "rarely". The findings were reported by project managers or members of upper-management.

6.3.3.2 Continuous Integration

Continuous integration, also, is one of the practices of XP. Its central idea is that the program code is integrated and tested very frequently – after few hours and at least daily [72].

In the respondent projects 43.7 % of the projects reported regular use of continuous integration (28.1 % systematically and 15.6 % mostly). However, nearly twice as many respondents (81.3 %) evaluated this practice as at least useful (34.4 % useful, 9.4 % very useful, and 37.5 % extremely useful). Clearly, also continuous integration is one of the agile practices that are evaluated as highly potential yet not used as much as might be required.

In addition, 18.8 % of the projects stated using continuous integration sometimes, 9.4 % rarely, and 21.9 % never. Two (6.3 %) of the respondents did not know the current usage of continuous integration practice in the project. Also, controversial opinions were found on usefulness of applying continuous integration. One respondent (3.1. %) regarded it as not useful, and one as harmful. However, either one of these projects had never applied the practice during this particular project, atleast. One of the respondents found continuous integration as not applicable in the project. Three of the respondents could not evaluate the usefulness of continuous integration (i.e., 9.4 % on "I do not know"). Three data points were missing on both the current use and usefulness of continuous integration.

6.3.3.3 Collective Code Ownership

"Anybody who sees an opportunity to add value to any portion of the code is required to do so at any time" is the definition of collective code ownership of Kent Beck [72]. It includes the basic idea that everyone is responsible for the whole of the system.

In the questionnaire study, the regular use of collective code ownership was reported in 42.4 % of the cases (18.2 % systematically, and 24.2 % mostly throughout the project). One project (3.0 %) reported sometimes using this practice and nearly one-third (30.3 %) of the respondents had never used it during the project. One respondent also regarded collective code ownership as not applicable and one did not have knowledge on this issue. 33 data points out of 35 were available on this topic.

[7] Please, note that in three of the questionnaires this data was not available.

The rating of usefulness was high in also this agile practice. As much as 78.1 % of the respondents reported that they consider collective code ownership at least useful (31.3 & useful, 28.1 % very useful, and 18.8 % extremely useful). As much as 12.5 % of the respondents could not evaluate how useful this practice would be in use ("I do not know") whereas only one respondent (3.1 %) regarded it as not useful, one as harmful and one as not applicable.

6.3.4 Quality Assurance Techniques

6.3.4.1 Test Driven Development (TDD)

Test-driven development focuses on writing and automating unit tests before the actual program code is written. It aims for verifying the functionality of the code for one, but also aims for guiding the development to a cleaner design and for giving developers confidence that the software works [76]. It is one of the practices adopted by XP [72].

In this questionnaire study, 18.8 % of the respondent projects (32/35) reported a regular use of TDD (i.e., 9.4 % systematically and 9.4 % mostly). As much as 40.6 % of the projects claimed of never using the approach during the project (i.e., 13/32). Two of the respondents (6.3 %) reported that TDD is not applicable in the project.

Due to the low current usage of TDD it can be said that the usefulness rate is largely based on the evaluated potential of TDD. As much as 71.9 % of the respondents evaluated TDD as at least useful to adopt in their project (i.e., 12.5 % useful, 31.3 % very useful, and 28.1 % very useful). There were also respondents with different opinions. Two respondents regarded TDD as harmful. Other one of these projects had sometimes applied TDD where as the other project had never applied it. Two respondents also reported TDD as not applicable and two of the respondents did not have an opinion (i.e., "I do not know"). Three data points were missing on this issue.

6.3.4.2 Refactoring

Refactoring is one of the XP practices. Its main idea to make the code as simple as possible while still running all the test cases. The programmer should work in two ways: 1) to enquire if the existing program could be changed to make adding the feature simple and 2) to enquire - after adding a new feature – if the existing program could be made more simple [72].

In the questionnaire study, it was revealed that nearly one third (30.3 %) of the respondent projects claimed having used refactoring either mostly (15.2 %) or systematically (15.2 %). Also, one third (33.3 %) of the projects (11/33) reported using refactoring sometimes in the project. Four of the respondents did not know the situation of using refactoring (12.1) and two claimed it to be not applicable (6.1 %). 18.2 % of the projects applied refactoring rarely (3/33) or never (3/33). Two of the respondents did not provide any information on this topic.

Also the (potential) usefulness of using refactoring was ranked high in the questionnaire study. As much as 75.0 % of the respondents (24/32) regarded using refactoring at least useful (15.6 % useful, 40.6 % very useful, and 18.8 % extremely useful). Again, these numbers exceed the reported level of use of refactoring currently. Thus, it can be argued that refactoring is seen as very potential practice that is not, currently, used as much as the software developers and project managers would be willing to. However, as much as 6.3 % (2/32) of the respondents found refactoring not useful. In Other one of these resopondents reported rare current use of refactoring in a project and another respondent did not know the current use refactoring in the project at hand. A total of 32 data points were available on the 35 questionnaires.

6.4 Future Research Needs and Limitations of the Study

One of the restrictions of the study was the convenience sampling that was used. In other words, the questionnaire was focused on a limited group of European software development organizations that are, assumably, active and interested in adopting agile practices (i.e., participants of Agile ITEA project). However, the target groups of this study were the project managers and software developers of these organizations. Assumably, the respondents are not directly involved in the Agile ITEA project and its goals in their daily work. However, it could be assumed that the results of this study may be favorable to agile practices compared to an “average” software development context. Thus, a future research may include an extended enquiry directed to a wider and independent variety of software engineering companies. It would be interesting to compare the results of these two questionnaire studies for finding out the differences in attitudes in different contexts but also to reveal how the adopting of agile practices will alter in few years time.

In the study, it was enquired how a specific project – either finished or ongoing – had been used a certain agile related method or practice. Also, it was acquired how useful the same method is/would be regarded. Based on the questionnaire, it was not possible to evaluate if the reported usefulness level was based on a real experience of the method (e.g., in some earlier project) or was merely an assumption. Only the responses where a certain method had been used mostly or systematically in the reported project it was clear that the usefulness was based on this knowledge. Also, it should be noted that the “usefulness” of a certain practice or method was merely a subjective matter of each respondent. The questionnaire did not provide any guidance on if the usefulness should have been evaluated, e.g., from the viewpoint of efficiency, product quality, or convenience. Rather, it should give some indication on how willing the respondent is in applying and adopting the practice.

Currently the agile practices, for example in XP, are designed to smallish teams working in co-located environment. One of the future research needs is to explore the potential and usage of agile practices in multi-site software development of embedded software.

6.5 Summary and Conclusions

In conclusion, it can be said that all the analyzed agile practices seem to be more appreciated than used currently in software development organizations. Based on this study, however, it is hard to evaluate a reason for this situation. One of the reasons might be the high publicity of agile software development yet also lack of guidance, expertise and effort on applying the new methods and techniques in more traditional context of software development. However, this study reveals that there is willingness on increasingly adopt new agile practices in software development organization and high expectations for their usefulness.

7. REFERENCES

[1] B. Boehm and R. Turner, "Using risk to balance agile and plan-driven methods," *IEEE Computer*, vol. 36, pp. 57-66, 2003.
[2] S. Ambler, *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons, Inc. New York, 2002.
[3] K. Beck, *Extreme Programming Explained: Embrace Change*: Addison-Wesley, 1999.
[4] K. Schwaber, "Scrum Development Process," presented at OOPSLA'95 Workshop on Business Object Design and Implementation, 1995.
[5] K. Schwaber and M. Beedle, *Agile Software Development With Scrum*. Upper Saddle River, NJ: Prentice-Hall, 2002.
[6] S. R. Palmer and J. M. Felsing, *A Practical Guide to Feature-Driven Development*, 2002.
[7] J. A. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York, NY: Dorset House Publishing, 2000.
[8] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*: Addison-Wesley, 2003.
[9] A. Cockburn, *Agile Software Development*. Boston: Addison-Wesley, 2002.
[10] J. Stapleton, *Dynamic systems development method - The method in practice*: Addison Wesley, 1997.
[11] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile Software Development Methods: Review and Analysis," VTT, Espoo 478, 2002.
[12] D. Cohen, M. Lindvall, and P. Costa, "Agile Software Development," 2003.
[13] S. Hayes, "Why Use Agile Methods?." Melbourne, Australia: Khatovar Technology, 2003.
[14] "Agile Methodologies Survey Results," Shine Technologies, 2003.
[15] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.
[16] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, and J. Zettel, *Component-based Product Line Engineering with UML*. New York: Addison-Wesley, 2001.
[17] "Component Oriented Software Manufacturing (COSM) methodology."
[18] D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Reading, Massachusetts: Addison Wesley, 1999.
[19] M. Awad, J. Kuusela, and J. Ziegler, *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and FUSION*. New Jersey: Prentice-Hall Inc., 1996.
[20] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons, 1994.
[21] "Developing Real-time Software With Rational Rose RealTime - version 6.0," Rational University 1999.
[22] A. Moore and N. Cooling, *Developing Real-time Systems using Object Technology*: Artisan White Paper, 2000.
[23] B. P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. New York: Addison-Wesley, 1999.
[24] "Telelogic Tau methodology web pages," 2005.
[25] "Object Management Group."
[26] J. Stankovic, "Strategic directions in real-time and embedded systems," *ACM Computing Surveys (CSUR)*, vol. 28, pp. 751 - 763, 1996.
[27] S. Van Baelen, J. Gorinsek, and A. Wils, "The DESS Methodology," 2001, pp. ITEA Project Report, DESS Deliverable D1.
[28] P. Kaiser and S. Van Baelen, "The EMPRESS Process," ITEA, ITEA Project Report December 2003 2003.
[29] "Model Driven Architecture (MDA)," Object Management Group July 9, 2001 2001.
[30] D. S. Frankel, *Model Driven Architecture, Applying MDA to Enterprise Computing*: OMG Press, 2003.
[31] A. e. a. Kleppe, *MDA Explained*: Addison-Wesley, 2003.

[32] L. Vandormael, "Common Characteristics with Attributes and Metrics," ITEA, ITEA Project Report February 2001 2001.
[33] J. Grenning, "Extreme Programming and Embedded Software Development," presented at Embedded Systems Conference 2002, Chicago, 2002.
[34] A. Pnueli, "Embedded Systems: Challenges in Specification and Verification (An extended abstract)," presented at Second International Conference on Embedded Software EMSOFT 2002, 2002.
[35] J. Ronkainen and P. Abrahamsson, "Software Development Under Stringent Hardware Constraints: Do Agile Methods Have a Chance," presented at XP 2003, Genova, Italy, 2003.
[36] IEEE, *Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)*: Institute of Electrical and Electronics Engineers, Inc., 1990.
[37] J. Taramaa, *Practical development of software configuration management for embedded systems*: Technical Research Centre of Finland, VTT Publications, 1998.
[38] J. Stankovic, "Real-time and embedded systems," *ACM Computing Surveys (CSUR)*, vol. 28, pp. 205-208, 1996.
[39] M. Mäkäräinen, *Software change management processes in the development of embedded software*: Technical Research Centre of Finland, VTT Publications 416, 2000.
[40] D. Dahlby, "Applying Agile Methods to Embedded Systems Development," 2004.
[41] B. Greene, "Using Agile Methods to Validate Firmware," *Agile Times*, vol. 4, pp. 79 - 81, 2004.
[42] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based Design and Software Design Methodology for Embedded Systems," *IEEE Design & Test of Computers*, vol. 18, pp. 23-33, 2001.
[43] M. Vierimaa, T. Kaikkonen, M. Oivo, and M. Moberg, "Experiences of practical process improvement," *Embedded Systems Programming Europe*, vol. 2, pp. 10 - 20, 1998.
[44] M. Borger, T. Baier, F. Wienberg, and W. Lamersdorf, *Extreme Modeling, Extreme Programming Examined*: Addison Wesley, 2001.
[45] J. Smith, "A Comparison of RUP and XP," *Rational Software White Paper*, 2001.
[46] G. Pollice, "Using the Rational Unified Process for Small Projects: Expanding Upon eXtreme Programming," *Rational Software White Paper*, 2001.
[47] P. Letelier, J. H. Canos, and E. A. Sanchez, *An Experiment Working With RUP and XP, Extreme Programming and Agile Processes in Software Engineering*: Springer, 2003.
[48] G. Booch, R. C. Martin, and J. Newkirk, "The Process," 1998.
[49] S. W. Ambler, "Are You Ready for MDA?," in *Software Development's Agile Modeling Newsletter*, 2004.
[50] S. W. Ambler, *The Object Primer: Agile Model-Driven Development with UML 2.0*: Cambridge University Press, 2004.
[51] Z. Stojanovic, A. Dahanayake, and H. Sol, *Component-Oriented Agile Software Development, Extreme Programming and Agile Processes in Software Engineering*: Springer, 2003.
[52] J. Martinsson, *Maturing XP through the CMM, Extreme Programming and Agile Processes in Software Engineering*: Springer, 2003.
[53] C. Larman, *Agile & Iterative Development: A Manager's Guide*: Addison-Wesley, 2004.
[54] C. Gelowitz, I. Sloman, L. Benedicenti, and R. Paranjape, *Real-Time Extreme Programming, Extreme Programming and Agile Processes in Software Engineering*: Springer, 2003.
[55] P. Kettunen and M. Laanti, "How to steer an embedded software project: tactics for selecting the software process model," *Information and Software Technology*, pp. 1-22, 2004.
[56] N. Shehabuddeen, F. Hunt, and D. Probert, "Insights into embedded software sourcing decisions: practical concerns and business perspectives," presented at IEEE International Engineering Management Conference IEMC'02, 2002.
[57] B. Greene, "Agile Methods Applied to Embedded Firmware Development," presented at Agile Development Conference, Salt Lake City, USA, 2004.
[58] K. Beck, M. Beedle, A. Bennekum van, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for Agile Software Development," vol. 2004, 2001.

[59] N. Van Schooenderwoert and R. Morsicato, "Taming the Embedded Tiger - Agile Test Techniques for Embedded Software," presented at Agile Development Conference, Salt Lake City, USA, 2004.
[60] R. Morsicato and M. Poppendieck, "XP in a Safety-Critical Environment," *Cutter IT Journal*, vol. 15, 2002.
[61] N. Van Schooenderwoert and R. Morsicato, "Freeing the Slave with Two Masters: An Embedded Programming Team's Transition to XP," *Cutter IT Journal*, vol. 15, pp. 34 - 41, 2002.
[62] D. Smigelschi, "Combining Predictability with Extreme Programming in Embedded Multimedia Project," presented at XP 2002, Sardinia, Italy, 2002.
[63] G. Mueller and J. Borzuchowski, "Extreme Embedded a Report from the Front Line," presented at 17th Annual ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'02), Seattle, USA, 2002.
[64] J. Bowers, J. May, E. Melander, M. Baarman, and A. Ayoob, "Tailoring XP for Large System Mission Critical Software Development," presented at XP/Agile Universe, Chicago, USA, 2002.
[65] D. Pierce, "Extreme Programming without Fear," *Embedded Systems Programming*, vol. 17, 2004.
[66] D. Turk, R. France, and B. Rumpe, "Limitations of Agile Software Processes," presented at 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002), Sardinia, Italy, 2002.
[67] N. Van Schooenderwoert, "Embedded Extreme Programming: An Experience Report," presented at Embedded Systems Conference, Boston, MA, 2004.
[68] J. E. Hewson, "Cross-Functional Pair Programming," *Embedded Systems Programming*, vol. 17, 2004.
[69] D. Pierce, "Agile Embedded: The Ground Floor," *Agile Times*, vol. 4, pp. 69 - 72, 2004.
[70] P. Manhart and K. Schneider, "Breaking the Ice for Agile Development of Embedded Software: An Industry Experience Report," presented at 26th International Conference on Software Engineering, Scotland, UK, 2004.
[71] "AGILE-ITEA web pages," 2004.
[72] K. Beck, *Extreme Programming Explained: Embrace Change*: Addison Wesley Longman, Inc., 2000.
[73] J. Highsmith, *Agile Project Management*: Addison-Wesley, 2004.
[74] H. Hulkko, "Pair Programming and its Impact on Software Quality," in *Department of Electrical and Information Engineering*: Unversity of Oulu, 2004, pp. 96.
[75] S. Heiberg, U. Puus, P. Salumaa, and A. Seeba, "Pair-Programming Effect on Developers Productivity," presented at XP2003, Genova, Italy, 2003.
[76] K. Beck, *Test-Driven Development By Example*, 1st ed: Addison-Wesley, 2003.
[77] N. Van Schooenderwoert, "Transition to XP in an Embedded Environment," *Agile Times*, vol. 4, pp. 72 - 74, 2004.

APPENDIX A: SUMMARY OF EMPIRICAL STUDIES

This appendix contains a summary of experiences on adopting agile practices in projects developing embedded software.

Reference	Application context	Motivation	Adopted agile practices	Experiences/findings
[70]	- Automotive industry (bus software)	- Constant changes in requirements - Critical time pressure - Need for high quality - Need for custom-made solutions	- Unit testing - Test first	**TDD:** - Previously used manual checking made it easy to separate typos from actual bugs - Automated testing requires more accurate syntax use and timing - Test cases need to be written more accurately and at a detailed level than before **Agile methods:** - No method can be adopted as-is - Step-by-step adoption required to minimize risk and gain acceptance - Effects of changes should be measured with a traditional measurement framework
[41, 57]	- Processor firmware dev. - 7 developer team - Code: 300 KLOC Itanium assembly and 30 KLOC C	- Many defects went undetected through testing - Constantly changing HW design in early stages of the project, because SW changes are cheaper - System fixing and tuning made with firmware after HW ready - Schedules could not be followed - Poor maintainability due to HW fixes and use of assembly - Too specialized team members, i.e. need for cross-training	**Scrum:** - Sprints - Sprint planning meeting - Daily Scrum - Sprint review **XP:** - Simple design - Unit testing - Refactoring - Pair programming - Collective ownership - Continuous integration - On-site customer - Sustainable pace - Coding standards	**Challenges for agile development:** - Firmware developers not keen on testing, because firmware changes are easier to make than HW changes - Parallel development of SW and HW -> critical firmware sections need to be tested regularly - Assembly code is more error-prone and needs low-level code implementation testing more **Simple design:** - Non-critical assembly code should not be optimized (causes poor maintainability) **TDD:** - Team had to develop their own unit test tool for assembler language - Positive results: increased quality, ease of coding, more focused planning and design, and timely feedback **Unit tests:** - Most useful practice - Enabled testing components at a low level (which is needed especially in assembly) **Refactoring:** - Reduced the amount of “kludginess” in the code - Improved the quality of the legacy code **Pair programming:** - Most controversial practice - In assembly coding, PP has value only in initial stages(detailed design, initial coding) - Cross-training benefits not as extensive as hoped, because the need for specialized domain knowledge in embedded system development too vast for anyone to have

Reference	Application context	Motivation	Adopted agile practices	Experiences/findings
[59, 61, 67, 77]	- Embedded real-time project - Product: mobile spectrometer - 3-year project, XP started after 1.5 years - 4 experienced team members - Code: 30 KLOC C and some assembly - Dual target RTOS	- Constant changes in algorithm design as it was in preliminary state - New project, no existing problems -Spiral development model, already some XP practices used: -- Collective ownership -- Coding standards -- Metaphor -- Continuous integration	- Planning Game - Pair programming - Test first - Simple design - Refactoring - 40-hour week - "Embedded TDD" (= TDD, trouble log, dual targeting, HW-related unit tests, domain-level tests, domain data tests)	**Planning game:** - Simplified the team's interface with multiple stakeholders (partner company, internal management, internal and external customers, mechanical and HW engineers) **Team meetings:** - half hour team meetings twice per week - additional meetings for longer topics as needed **Pair programming:** - Very beneficial - Also code reviews performed, which became more valuable and active, as everyone new the code they were reviewing **Simple design:** -An initial overall architecture, developed prior to the adoption of XP, remained useful -Up-front designing: --in the beginning of each iteration, the design was revised due to the features that were added -- designing for the current iteration and making some provisions for the one after that -- detailed designing in order to have valid estimates --designs were analyzed using several use case scenarios. - HW-related issues were planned as far as possible - Time period between releases narrowed towards the end of the project because system-level problems were fixed with software changes **Refactoring:** - Before XP, root cause of most bugs was code reviews, and after, insufficient refactoring - Checklists were created and used as needed - Code reviews for code that wasn't pair programmed **Documentation:** - the first half (before XP): a software design document at the start of each planned "spiral" - the second half: -- XP prompted to ask the following questions about the software design document at the start of each iteration: "Is this necessary now" or "What is the simplest thing that could possibly work" - Brief developer-level documents, "Mini-docs", were created as needed, and maintained by the project team. **Test first:** - Easy to adopt - Each module's test program was compiled and ran every night **Unit tests:** - Trouble log which used little memory and was fast to execute was used to obtain error descriptions - Dual targeting (i.e. ability to run the SW on both target HW and PC) used to separate SW and HW bugs, and to enable automation of unit tests - HW-related unit tests used to test HW controllers and drivers independently during development, and by HW vendor as final test suite before shipping **System level tests:** - Domain tests were used to test separate domains, when unit testing was too fine-grained and system testing too high-level. One domain was built and loaded to HW, and I/O from rest of the system replaced by mock objects. This was used e.g. for tracing timing problems - Domain data tests, i.e. testing the SW with design data, used to keep up with constantly changing requirements and verify the new code against them **Overall TDD results:** - Low bug rate - System bugs not instantly blamed on SW, because it was easily testable

Reference	Application context	Motivation	Adopted agile practices	Experiences/findings
				- Better communication between HW and SW engineers **Overall software process** - the first half: "generic agile", spiral development model - the second half: "Embedded XP" **"Generic agile"** - unit tests foe every module but no automatic unit tests - collective ownership of code - iterative releases - no pair programming but code reviews - no planning game **Adoption of XP: "Embedded XP"** -: Dual targeting and trouble log perceived as helpful practices which supported the adoption of XP practices - Successful transition to XP enabled by the fact that already a subset of XP practices was used in the project - Testability of the system is a key issue when transitioning to XP - "Now we really understand how interdependent the XP practices are – if you're going to use it, you should use all of the practices."
[62]	- Automotive multimedia project	- Low staff motivation - Unclear requirements - Bad customer communication - Inefficient QA and documentation	**Agile-like practices:** - Team co-location based on similar activity types - Defining clear roles for everyone - Bi-weekly status meetings with customer - Appointing project members responsible for customer communication	**Customer communication:** - Improved visibility, enhanced attitude - Clear definition of expectations **Planning:** - Estimations were done together with team members to increase commitment - Stable and volatile parts of SW were identified, and long term planning was made to the stable ones, but the volatile parts were planned only one iteration ahead
[63]	- Embedded legacy product development - 10 years old product - 9 programmers with >10 years of experience - 100 KLOC C code	Waterfall process used - Vague or missing requirements - Over 2000 page design document - Bad intelligibility and maintainability of the SW - No separate SW testing was used - Complex and time-taking integration	- Pair Programming - Stories - Unit testing - Refactoring - Metaphor - Continuous integration	**Pair programming:** - Experienced paired with inexperienced to propagate product knowledge - Same pairs used to complete a task - Pairs worked on design and complex coding tasks, simple and repetitive tasks were done solo in a co-located office space - Co-ordination was needed to match pair member's working times **Continuous integration:** - The used tool built new code base slowly and locked the code until the integration was ready, so developers did not integrate frequently **Unit testing** - Not mandatory, so many pairs did not write them - Existing tests were not updated according to code changes, so there was nothing to test refactorings against **Customer:** - End user not interested from software or a part of software, but from the whole embedded product, and thus getting priorities from the end customer is not feasible - Hardware team used as customer to define and prioritize requirements **Coding standards:** - In the legacy product's code, many different coding styles were used. - Not feasible to impose one coding standard to be followed. - Agreed, that changes to a module would be done in the

Reference	Application context	Motivation	Adopted agile practices	Experiences/findings
				same style as the rest of the module - For a new module, one of existing styles were chosen **Adoption of agile methods:** - Embedded SW developers are more sceptic and resistant to changes because of their HW backgrounds - XP coach and upper management commitment needed to succeed
[64]	- Re-implementation of a part of a legacy SW product - Mission-critical SW - Soft real-time requirements - Over million LOC of C code	- Aim to mitigate risks with early, frequent feedback - One of several pilot projects at a large company	Several XP practices: - Small releases - Continuous integration - Pair programming - Simple design - Planning Game - Refactoring - Testing	**Small releases:** - 3-month release cycle chosen so that one feature could be done in one iteration - Release schedule was integrated to system integration and testing schedules - Because development was focused at one feature at a time, feature's interaction was not considered enough and thus the most complex tasks were faced at the end of the project **Continuous integration:** - Integration to mainline controlled by a bi-weekly change control board (CCB) - CCB accepted changes and assured that the change requests had been fulfilled - Longer integration enabled to think about the effects of a change in detail, but made the integration harder and more time taking **Pair programming:** - High initial adoption, which decreased during the project, and finally, only risky changes were made in pairs - Formal reviews omitted at first, but then brought back due to low usage of PP, missed defects and organizational pressure - In mission-critical environment, PP should be accompanied by formal code and test case reviews - In complex environments, experts and feature owners are needed over common knowledge (knowledge transfer benefits of PP not as extensive or even important) **Simple design:** - In mission-critical environment, requirements rarely change because they have been diligently defined in a contract between the customer and business team prior to the project's inception -> 80-90% of all requirements should be completely defined prior to the first iteration - If more time had been spent on design, high-risk areas of the code and interaction of the features might have been identified better **Planning Game:** - No access to end customer, but expert from system design group worked as the customer, which created a conflict of interest - Only 80-90% of the requirements were captured at planning game, rest were discovered during implementation (due to lack of end customer) - If dedicated customer is not an option, majority of requirements should be established before the first iteration **Refactoring:** - Permission for refactoring had to be asked from the CCB (due to mission-criticality) - Large refactorings created significant defects (too much reliance on test suite to detect possible defects caused by refactoring) **TDD:** - 1500 unit test written and merged into a test suite - Testing perceived very useful and critical to refactoring - Too much reliance on the test suite, a balance between code reviews, automated regression tests and

Reference	Application context	Motivation	Adopted agile practices	Experiences/findings
				acceptance tests should be used - Large test suite takes a lot of effort to maintain according to changes -> Sometimes design decisions were affected by unwillingness to make major changes to the test suite - Also a test suite for testing interfaces with legacy code would have been necessary **Overall results:** - Positive overall experiences - Product quality well within expectations
[60]	- Pharmaceutical instrument - Safety-critical	- Aim was to find an effective approach to help to identify as much failure scenarios as possible throughout the project, enable their prevention and yet be able to control and trace the effects of new changes and corrections	- Refactoring - Unit testing - Stories	**Stories:** - Requirements traceability which is important in safety-critical development, results from XP, since no code is written if there's no user story requiring it **Refactoring:** - Refactoring a design is good for discovering failure scenarios and finding design flaws - Root-cause analysis of bugs revealed, that the team should have done refactoring when they felt like it **Testing:** - Perceived as the most important discipline of XP - Emphasis should be put on rigorous unit testing to produce better and safer code - XP's testing practices help to assure, that the safety controls are not violated when the system is changed **Overall results:** - XP practices are adopted more voluntarily than "traditional" processes, because XP contributes and improves the developers work directly, and does not feel like an extra burden forced by the process improvement instances - XP process did not pass an audit, although it was regarded beneficial and efficient