# Customized agile development process for embedded software development

UNIVERSITY OF TURKU
Department of Information Technology
Master of Science in Technology Thesis
June 2010
Tomi Juhola

Inspectors:
Tuomas Mäkilä
Ville Leppänen

UNIVERSITY OF TURKU
Department of Information Technology

TOMI JUHOLA: Customized agile development process for embedded software development

Master of Science in Technology Thesis, 106 p.
Software Engineering
June 2010

---

Agile software development has grown in popularity starting from the agile manifesto declared in 2001. However there is a strong belief that the agile methods are not suitable for embedded, critical or real-time software development, even though multiple studies and cases show differently. This thesis will present a custom agile process that can be used in embedded software development.

The reasons for presumed unfitness of agile methods in embedded software development have mainly based on the feeling of these methods providing no real control, no strict discipline and less rigor engineering practices. One starting point is to provide a light process with disciplined approach to the embedded software development.

Agile software development has gained popularity due to the fact that there are still big issues in software development as a whole. Projects fail due to schedule slips, budget surpassing or failing to meet the business needs. This does not change when talking about embedded software development. These issues are still valid, with multiple new ones rising from the quite complex and hard domain the embedded software developers work in. These issues are another starting point for this thesis.

The thesis is based heavily on Feature Driven Development, a software development methodology that can be seen as a runner up to the most popular agile methodologies. The FDD as such is quite process oriented and is lacking few practices considered commonly as extremely important in agile development methodologies. In order for FDD to gain acceptance in the software development community it needs to be modified and enhanced.

This thesis presents an improved custom agile process that can be used in embedded software development projects with size varying from 10 to 500 persons. This process is based on Feature Driven Development and by suitable parts to Extreme Programming, Scrum and Agile Modeling. Finally this thesis will present how the new process responds to the common issues in the embedded software development.

The process of creating the new process is evaluated at the retrospective and guidelines for such process creation work are introduced. These emphasize the agility also in the process development through early and frequent deliveries and the team work needed to create suitable process.

Keywords: agile software development, Feature-Driven Development, Agile Modeling, embedded software development, software development process

TURUN YLIOPISTO
Informaatioteknologian laitos

TOMI JUHOLA: Räätälöity ketterä ohjelmistokehitysprosessi sulautetun ohjelmiston kehittämiseen

Diplomityö, 106 s.
Ohjelmistotekniikka
Kesäkuu 2010

---

Ketterä ohjelmistokehitys on kasvattanut suosiotaan ketterän manifestin julkaisusta vuonna 2001 lähtien. Tästä huolimatta yhä uskotaan, että ketterät menetelmät eivät ole sopivia sulautettujen, kriittisten tai tosiaikaisten ohjelmistojen kehittämiseen, vaikka useat tutkimukset ja tapaukset ovat todenneet toisin. Tämä opinnäyte esittelee räätälöidyn ketterän prosessin, jota voi käyttää sulautetun ohjelmiston kehittämiseen.

Ketterien menetelmien oletettuun sopimattomuuteen sulautettujen järjestelmien kehittämiseen on useita syitä, jotka perustuvat tuntemukseen, jonka mukaan menetelmät eivät tarjoa oikeaa kontrollia, tiukkaa kurinalaisuutta ja täsmällisiä kehityskäytäntöjä. Yksi lähtökohta on tarjota kevyt ja kurinalainen prosessi sulautettujen järjestelmien kehittämiseen.

Ketterä ohjelmistokehitys on saavuttanut suosiota, koska nykyisessä ohjelmistokehityksessä on suuria ongelmia. Projektit epäonnistuvat aikatauluhaasteiden, budjettiylitysten tai liiketoimintatarpeisiin sopimattomuuden vuoksi. Tilanne ei ole erilainen sulautettujen ohjelmistojen tapauksessa. Nämä ongelmat ovat edelleen valideja ja lisäksi monia muita ongelmia esiintyy monimutkaisesta ja vaikeasta alasta johtuen. Opinnäyte käyttää näitä ongelmia lähtökohtana prosessikehitykselle.

Opinnäyte perustuu voimakkaasti Feature Driven Development menetelmään, joka on melko suosittu ketterä menetelmä. FDD on melko prosessiorientoitunut ja siitä puuttuu monia käytäntöjä, joita pidetään erittäin tärkeänä ketterille menetelmille. Jotta FDD saadaan paremmin käyttöön ohjelmistokehitysyhteisössä, sitä pitää muokata ja parannella.

Opinnäyte esittelee parannellun räätälöidyn ketterän prosessin, jota voidaan käyttää sulautettujen järjestelmien projekteihin, joiden koko voi vaihdella kymmenestä 500 henkilöön. Tämä prosessi perustuu Feature Driven Development menetelmään ja sisältää myös sopivia osia Extreme Programming, Scrum ja Agile Modeling menetelmistä. Opinnäyte näyttää myös kuinka uusi prosessi vastaa yleisimpiin sulautetun ohjelmistonkehityksen haasteisiin.

Uuden prosessin kehitysprosessi arvioidaan retrospektiivissä ja prosessikehitykseen esitetään ohjeita. Nämä ohjeet korostavat ketteryyttä myös prosessikehityksessä aikaisten ja jatkuvien toimitusten avulla sekä tiimityöskentelyn tärkeyttä painottaen.

Asiasanat: ketterä ohjelmistokehitys, Feature Driven Development, Agile Modeling, sulautetun ohjelmiston kehitys, ohjelmistokehitysprosessi

# Table of Contents

## TABLE OF FIGURES

# 1 Introduction

Agile software development has grown in popularity starting from the agile manifesto declared in 2001. However there is a strong belief that the agile methods are not suitable for embedded, critical or real-time software development, even though multiple studies and cases show differently. This thesis will present a custom agile process that can be used in embedded software development.

The reasons for presumed unfitness of agile methods in embedded software development have mainly based on the feeling of these methods providing no real control, no strict discipline and less rigor engineering practices. One starting point is to provide a light process with disciplined approach to the embedded software development.

Agile software development has gained popularity due to the fact that there are still big issues in software development as a whole. Projects fail due to schedule slips, budget surpassing or failing to meet the business needs. This does not change when talking about embedded software development. These issues are still valid, with multiple new ones rising from the quite complex and hard domain the embedded software developers work in. These issues are another starting point for this thesis.

The thesis is based heavily on Feature Driven Development, a software development methodology that can be seen as a runner up to the most popular agile methodologies. The FDD as such is quite process oriented and is lacking few practices considered commonly as extremely important in agile development methodologies. In order for FDD to gain acceptance in the software development community it needs to be modified and enhanced.

The aim of this thesis is to present an improved custom agile process that can be used in embedded software development projects with size varying from 10 to 500 persons. This process is based on Feature Driven Development and by suitable parts to Extreme

Programming, Scrum and Agile Modeling. Finally this thesis will present how the new process responds to the common issues in the embedded software development.

The motivation for development of an own agile process was to be able to provide a suitable agile embedded software development process for customers alongside with training and consulting services. The process is targeted for a major customer in Finland that has a need for agile methods, but that is not sure how they will fit into their environment and how to adopt the process.

# 2 Agile development

This chapter describes what the agile development means and why it has become very popular during the last decade. The basis of agile software development, the Agile manifesto [Man01], is discussed also to describe the rationale of the values in agile manifesto and also defining how the values and principles correlate with each other.

After that the Feature Driven Development [Palm02] and Agile Modeling [Amb02] are presented in detail as they are important basis for the proposed process of this thesis. After this other agile software development methods are presented, including the popular Extreme programming [Beck99], [Beck05] and Scrum [Schw01], and also a rationale for choosing the Feature Driven Development is presented.

## *2.1 Agile Development*

Traditional software development relies on waterfall model [Roy70] based processes. These start from requirements gathering, moving forward to design phase and then finally at the end implementing and testing the system. These processes have tight procedures for work and tight requirements for delivered artifacts. Still very large number of software development projects fails. Standish Group has followed the state of software development continuously since they

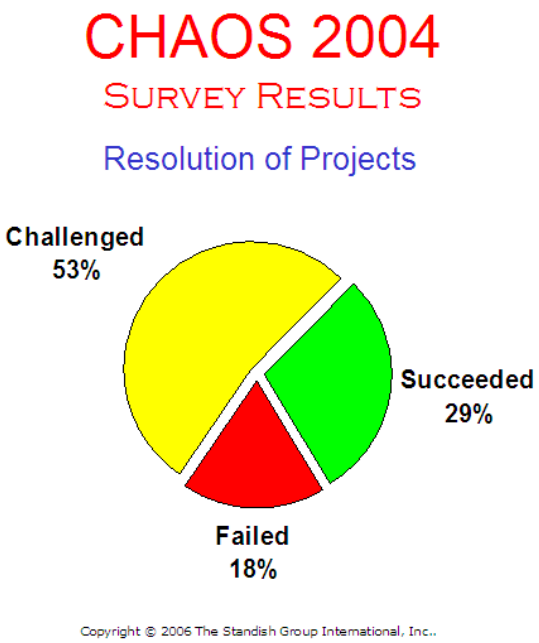


**Figure 1 CHAOS 2004 Survey results [Info06].**

published the first CHAOS report in 1994 [Sta94]. Standish group reports in the CHAOS report of 2004 that only 29 % of software projects are successful [Sta04] as displayed in Figure 1. Also Capers Jones has received similar results from his study spanning from 1995 to 2004 and concentrating on 250 large software projects [Jones04]. The study reports that 175 of the projects (70 %) experienced major delays and cost overruns.

Ian Sommerville in his book, “Software Engineering” [Som04], offers an explanation by pointing out that the traditional development methods and processes originate from large, critical, long-lived systems developed by distributed teams and composed from a large number of individual subprograms.

The traditional software development usually features a very detailed project planning, formalized quality assurance, many tools aiding analysis and design and, most of all, very controlled and tight software development processes. These cause some overhead and wasted effort in planning, designing and documenting, but Sommerville thinks that this overhead is justified when talking about very large systems and distributed development teams. However when the traditional software development methods are used in small or medium sized projects, the waste can have a big impact. This leads, according to Sommerville, on situations where more time is spent on how the system should be developed than on the development and testing itself. [Som04]

Sommerville thinks that the problems of traditional approach arise also from the markets of today. The software companies operate in a global, rapidly changing environment where they have to respond to new opportunities and markets. This leads to a situation where the complete set of stable requirements are usually impossible to deliver. The requirements change according to markets and new knowledge gained during development. Sometimes the real requirements can be found after delivery of the software from the feedback of customers and users. This leads to a situation where a project is delayed because of new requirements and this again leads to disappointed customers because of slipping schedules, growing budgets and, in worst cases, bad quality. [Som04]

Software is often compared to product manufacturing and from this metaphor comes often the reason for software development processes. In manufacturing the product quality comes from an improved process that is finally standardized to produce suitable quality products every time. However in software development the case is not the same from three reasons introduced by Ian Sommerville [Som04] and an additional reason by Mary and Tom Poppendieck [Poppen07]:

1. The specification should tell what the customer wants. However in software development there are requirements that come from development organization, e.g. maintainability, reusability [Som04].
2. Certain quality characteristics (e.g. maintainability, security) can not be or are extremely hard to specify in an unambiguous way [Som04].
3. It is extremely difficult or even impossible to write a complete specification in advance. Even though the software would perfectly fit the specification it might not suit the needs of the users [Som04].
4. Software development processes are subject to constant change and require learning from the developers [Poppen07].

The previous four reasons are also the main reasons when deciding what kind of process is used in product development. There are two different ways: The empirical process control and the defined process control (or planned process control) [Poppen07]. The empirical process starts with a high-level product concept, called e.g. a vision in Scrum [Schw01]. The concept is interpreted to a product by using well-defined feedback loops that adjust activities constantly [Poppen07]. The defined process control creates a complete product specification and delivers a product based on that according to a plan [Poppen07]. The reasons in previous paragraph should drive the software projects to select the empirical process as the way to develop software. Agile development methodologies are based on empirical process control and continuous feedback. This has also been used in many other creative areas of work and in different product development projects, e.g. Toyota uses empirical processes in development of new cars in their own Toyota Product Development System [Lik04].

The Agile development offers a cure for these problems, by focusing on the software itself. The principles of agile development are described in Agile Manifesto [Man01] that was published by a group of software developers believing in light-weight methodologies. There are many features that are common to most agile development methodologies. The methodologies are individual based, iterative, incremental, focusing on client valued functions and change adaptation.

The first basic principle is that the individuals and interactions are more important than processes and tools. This means that face-to-face real-time communication is emphasized and written documents should be produced only when needed. What comes to individuals, one of the Agile Manifesto signatories, Robert C. Martin describes [Mart03] that the professional goal of every software developer is to deliver the highest possible value to their employers and customers.

The incremental and iterative design starts from a small set of requirements that are best understood and have the highest priority and use those to develop working software with some client valued functions. The software is delivered or shown to customers and users and from the feedback new requirements are found. The real requirements come up in a very early stage of the project instead of the end of the project that was the case with traditional approach. This short step, or iteration, is repeated until the software is ready. The incremental development in a way reflects the fundamental way we all tend to solve problems. Taking a small subproblem at a time and solving that until the whole problem is solved [Mart98]. The difference in traditional iterative waterfall model and

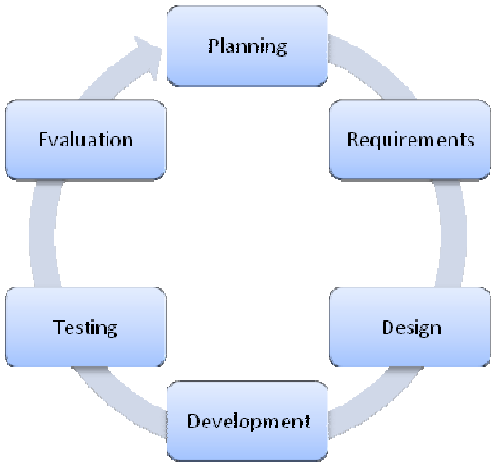


**Figure 2 Iterative and Incremental model.**

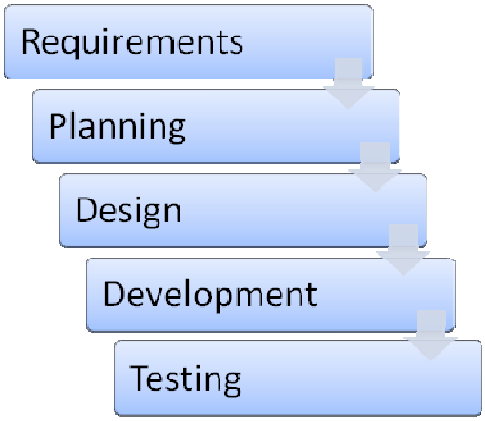


**Figure 3 Traditional waterfall model.**

the agile iterative and incremental model can be seen in process models (see Figures 2 and 3).

Focusing on client valued functions makes sure that the client gets as much value as possible from the project. Instead of delivering large buggy software with all the functions required, the agile developers deliver working software with the most important features. According to Jim Johnson, the Chairman of Standish Group, only 20 % of the requirements are always or often used in software projects. 45 % of the requirements are rarely or never used [John02]. The data is also displayed in Figure 4.

In Lean software development this is seen as one of the three biggest wastes in software development [Poppen03], [Poppen07].

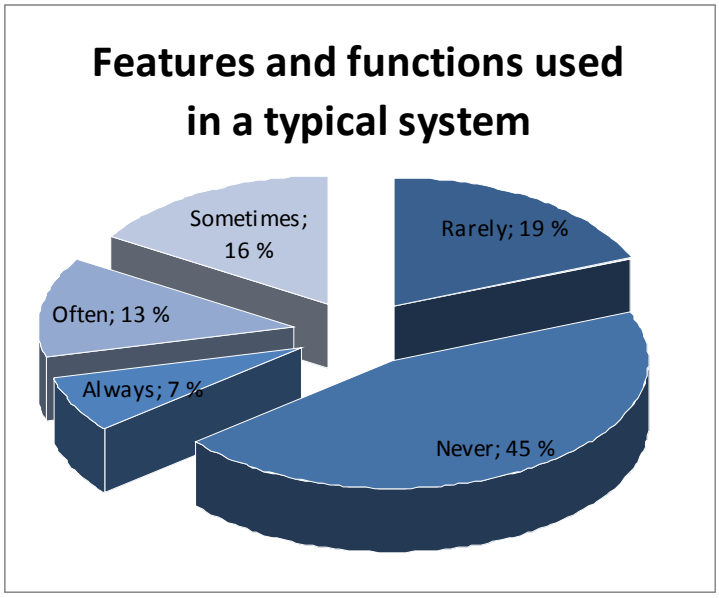


**Figure 4 Almost half of the features are never used [John02].**

In today's software projects there will be a large number of changes. The amount of changes has been estimated to be between 1 % to 3 % of requirements changes per calendar month and averaging to 25 % or more requirements changes per project [Jones97]. The agile development welcomes these changes. The changes do not make development as hard as it makes in traditional development because the changes come steadily through the project. In traditional development most of the changes seem appear at the final stages as the acceptance testing and high level functional testing are performed. At that time the responding to them is a lot harder. Incremental development makes implementing changes easier as the changes will not affect the whole system but just the increments of software already developed [Beck05].

The agile development was summarized by Andrew Hunt and Venkat Subramaniam with a one phrase: “Agile development uses feedback to make constant adjustment in a highly collaborative environment” [Sub06]. Kent Beck, the father of extreme programming (XP), compares the agile development and XP to driving a car on a curvy road; you have to make all the time some adjustments to reach your destination [Beck99].

The agile development has naturally its own problems and it has been criticized. Maybe the most well known criticism has come from Barry Boehm and Richard Turner, who suggest that risk analysis should be used to choose between agile (adaptive) and traditional plan-driven (predictive) methods [Boehm03]. The customer involvement is usually hard as the customers are used to giving requirements and making a fixed price contract on building the software according to those requirements. Customers are not ready to invest more time to the software development. The emphasizing of interactions and individuals is criticized as different people may have a hard time working and communicating together. So bringing face-to-face communication as a main way to communicate makes some individuals uneasy. As the agile development seeks always the simplest possible solutions this may lead to a situation where a lot of extra work is needed in order to maintain that simplicity [Mart03].

On the management level the agility can cause some contract problems. As mentioned before, the industry is used to give requirements and receive working software more or less on time. The agile way of development requires more feedback and more communication from the customer’s side. This is something that is not easy to explain to customers and to make the customers also committed to the project. The agile promise to the customers is to bring the developers closer to the customers and help them in the following issues [Poppen03b]:

- What is the simplest way to address the customer's business problem?
- How to best deliver what is needed?
- How to deal with changes over time?

Along these criticisms it is usually said that agile development fits best for small and medium sized projects and not for distributed development, critical system development or embedded systems development [Som04]. However the view on project size has changed and there are already articles on agile software development on large projects [Eck04], [Lef07], [Schw07] as well as case studies on the suitability of agile approach to such projects [Haa07].

The agile approach is said to be bad for embedded software development as the embedded software depends so much on the hardware and the final hardware may be unavailable until the final stages of the project. The process described in this thesis tries to answer some of the criticism before and develop a basic process that can be used as a basis for embedded systems development projects of varying size.

## *2.2 Agile Manifesto*

"Agile methods" is a common term for a family of light-weight development processes, not a single approach to software development. In 2001, 17 light-weight methodology experts gathered at the Snowbird ski resort in Utah to discuss the unifying of different light-weight processes. There were representatives from Extreme Programming [Beck99], Feature-Driven Development [Palmo02], SCRUM [Schw01], DSDM [DSDM03], Adaptive Software Development [High00], Crystal [Cock01], Pragmatic Programming [Hunt99], etc. All of the representatives agreed that there was a need for an alternative to traditional documentation driven, heavyweight software development processes. They created and signed the Agile Software Development Manifesto, which defines agile development, and accompanying 12 agile principles, guiding the software professionals. This meeting also gave the birth to Agile Alliance, a non-profit organization that supports individuals and organizations who use agile approaches to develop software. [Man01]

The significance of the manifesto in my opinion is that it created a tempting brand for the lightweight processes and described a practical and reasonable basis for agile

software development. This has certainly helped the increasing popularity of agile development amongst the software developers as the developers have been able to see the conflict with these values and the contradicting current values.

The manifesto promotes four items that should be valued: Individuals and interactions, working software, customer collaboration and responding to a change. These are treated as the most important pieces of software development projects. The works so that, the items on the right side in Figure 5 are very important but their importance should not exceed the items on the left side. Even if some people seem to misinterpret the manifesto, the items on the right side should not be forgotten, but they should not be used to create waste. [Man01]

**Manifesto for Agile Software Development**

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

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

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

**Figure 5 Agile Manifesto [Man01].**

The first value, individuals and interactions over processes and tools, reminds that the software is not created by just processes or tools, but that people are needed in order to make working and valuable software. The interaction of these individuals is often something that is forgotten as a key point in software development. In fact Tom De Marco and Timothy Listener stated some results of their researches in their book Peopleware [DeM87]: “The major problems of our work are not so much technological as sociological in nature". This is further interpreted by De Marco and Listener to mean that the one key success factor in software development is the ease and efficiency of communication between people.

The second value, working software over comprehensive documentation, seems like quite obvious. The aim is to create working software. Comprehensive documentation should not go ahead of working software. This value also creates a common misunderstanding that agile software development means that there is no documentation, but the real idea is that unnecessary documentation is avoided. Each created document should have value to the customer or some other stakeholders. Fred Brooks has identified few needed documentations for software in his essay "The Other Face" [Bro75]: User's manual to know how it is used, test cases for proving it works, a flow chart (or nowadays architecture description and class diagrams) to be able to modify it and a simple, commented and readable source code to define the software behavior more precisely. These can be thought as the needed, valuable documentation. When creating additional documentations, the value of that document should be considered before creating it.

Next value, customer collaboration over contract negotiations, is quite hard to follow in modern business environment. The aim is to make the customer happy, meet his needs, and create a tight cooperative relationship with him. This also emphasizes the value of long time customers. Instead of making as good and beneficial contracts as possible and trying to find holes in the contract to make the software easier to develop, the developers and the customer should work closely together to find out the customer’s needs and to build the best possible software in given time and budget frame to meet these needs. This is in fact mandatory way to define requirements, as it has been noted that there is no way for the customer to define the requirements in advance [Bro75]. This way also the customer’s confidence and satisfaction is build up day by day and the developers learn the customer’s domain area little by little.

The last value states that it is more important to respond to change than to follow a plan. Again some might say this is more common sense than software development. Software development is performed in quite turbulent environment where change is inevitable as stated before. The changes come from quite large number of sources: customers, users, management, developers, technology, business, risks coming true etc. When changes occur, we have to create a new plan and start using it, instead of sticking with the old

plan. The ability to respond to changes in timely manner makes the small companies successful; however this ability seems to be vanishing alongside the growth of the company and with more emphasis on more strict processes. Agile company is a company that can react to changes rapidly and the reaction is close to correct.

The values of agile manifesto guide the people involved in a software project to work in an agile manner. These values themselves are quite abstract and hard to implement in practice but they give the basis on how to solve problems and do really valuable work. Because of the abstractness, the agile manifesto also includes 12 principles. These give you an insight on what the agile values mean in more practical sense. These are also common principles for all agile development methodologies [Man01].

**Principles behind the Agile Manifesto**

*We follow these principles:*

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.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

**Figure 6 Principles of Agile Manifesto [Man01].**

The values and principles of agile manifesto correspond to each other according to our mapping in Figure 7. The principles are general guidelines that can be used to guide the development project through unusual situations. However the day-by-day work should follow the processes and practices described in the agile methods, as presented in Sections 2.3-2.6.

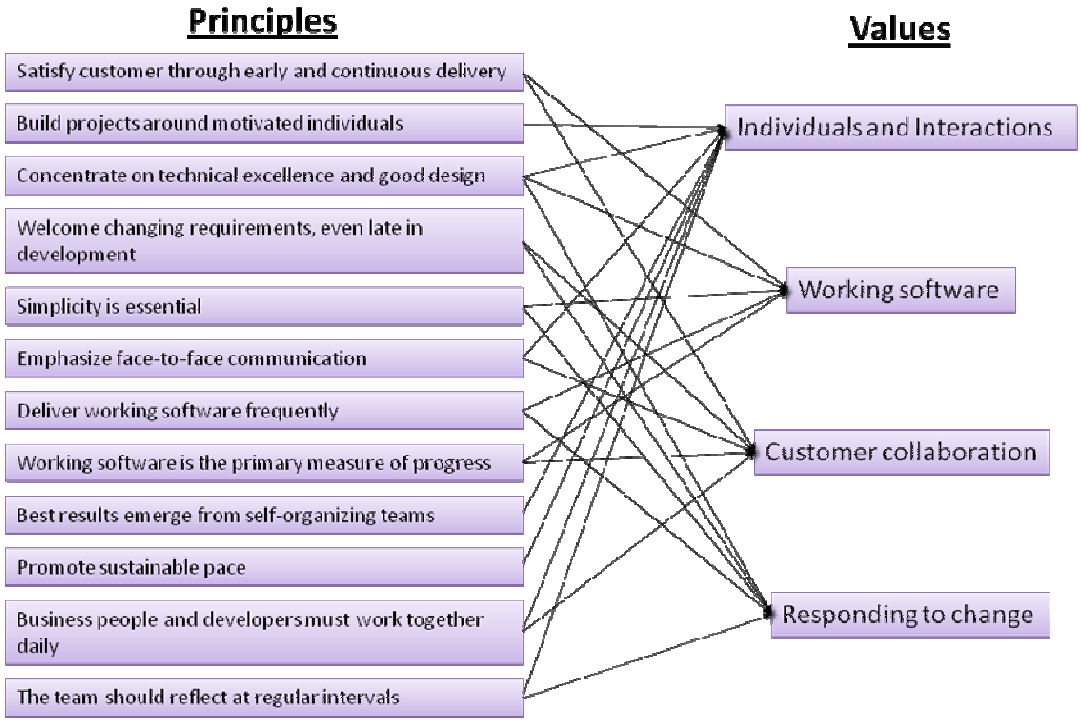


**Figure 7 Description how the principles map to the values.**

## *2.3 Feature Driven Development*

Feature Driven Development [Palm02] was developed by Jeff De Luca for a large banking software project in 1997. The original process was highly influenced by Peter Coad's ideas of object modeling, color modeling and development processes. It was in fact presented first time as a chapter of color modeling book [Coad99]. This original process was later on refined by Stephen R. Palmer and John M. Felsing [Palm02] to its current form, making it more general and decoupling it from the color modeling. FDD has since become one of the most popular agile software development methods [Amb06], however lately the trend seems to be towards Scrum and own hybrid agile models [Ver07].

Feature-Driven Development, or FDD, is an incremental, short-iterative and model-driven process consisting of five basic activities. These are [Palm02]:

- Develop an overall model
- Build a Feature List
- Plan by Feature
- Design by Feature
- Build by Feature.

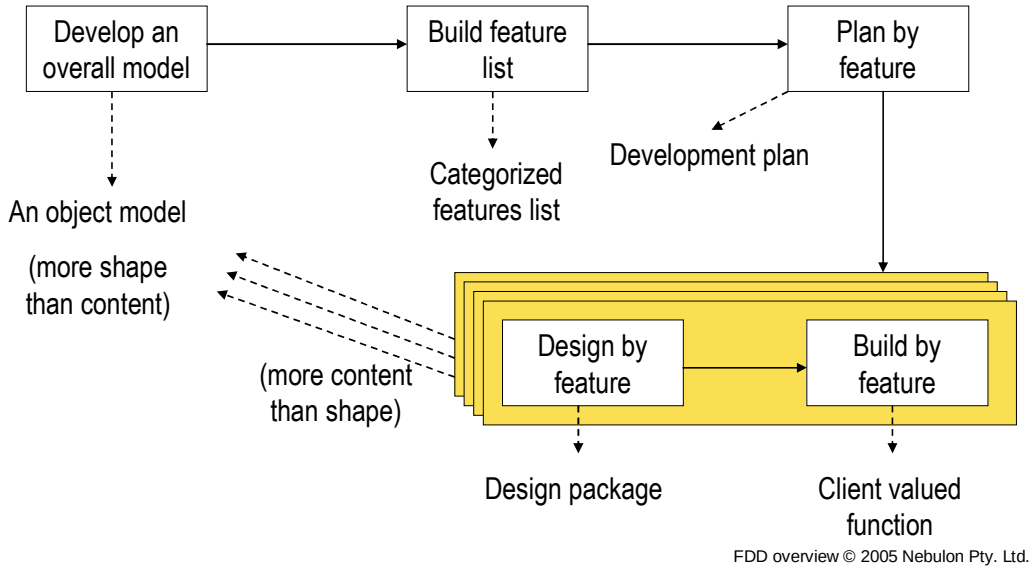


**Figure 8 Basic process flow of FDD [Nebu05].**

Develop an overall model is namely the first step of FDD based software developing although there may have been some prototyping and preliminary business related planning done, or even formal and traditional requirements gathering. Whatever the starting point is, the aim is to create, refine and add more detail to the model. In this phase the project team starts forming and gets more involved in project and the management has decided that at least first three phases are to be done. [Palm02]

Domain modeling starts with a domain walkthrough guided by domain expert. The modeling team then studies documents, if available, and develops models of the domain

in groups of three. The chief architect may give team a starting point in a form of “straw man” model. The models are then presented and discussed. The result of this is a model of domain that is updated to overall model. The domain model is accompanied with notes from modeling session that state some design decisions and explain more complex structures. Also alternative domain models should be documented. The model is verified internally by active participation of modeling team and, if needed, it is verified and clarified by customers and users. This phase is repeated to every domain in order to build a complete overall model with a suitable detail level for needs of the project. [Palm02]

The next step is to build a feature list. FDD uses features to communicate about requirements with the customer and to divide the project to simple, easy-to-implement parts. A very important part of the features is that they should be client-valued features. The features are created according to a simple template: <action> <result> <object> (e.g. “Track number of parking passes” and “Enter the desired number by dial pad”). The building of a feature list starts with identifying the set of features. These are decomposed into subject areas. As previously mentioned, the features should be client-valued functions that are granular enough to implement in short time (max. 2 weeks). Granularity should not also go too far, getters and setters do not need own features and the same goes with parts of UI. This phase, or subprocess, should result in list of subject areas consisting of business activities and features to complete those business activities. [Palm02]

The most important part of the first phase, plan by feature, is determining the development sequence of features. This is done by analyzing the complexity and the size of the feature and the dependencies it has to classes. The feature sets are assigned to chief programmers after this, who in turn assigns classes to developers. The verification is again done by self assessment of the team, and the business value and implementability should be considered during this phase in order to avoid starting implementation of a process too risky or hard. This phase ends the requirements gathering, design and analysis phase and studies the possibilities of project’s success [Palm02]. Also, as Barry Boehm stated: “Project termination does not equal project

failure" [Boehm00], but in fact it is many times more profitable to end unfeasible products before any more time, effort and money is used to it. This kind of projects can be found through prototyping or through e.g. risk and market analysis according to Boehm.

The next two phases of FDD are repeated iteratively in order to grow incrementally, feature-by-feature, quality software that meets the demands of customers. The third phase, design by feature, starts with forming a feature team by chief programmer for a chosen feature. If this feature is complicated, the feature team is introduced to domain of the feature by domain expert and the team may have to study some additional documents in order to be able to design the feature. The team then creates sequence diagrams that describe the actions of the feature. If new classes and attributes are found, the chief programmer should update these to overall object model. After this the class owners in the feature team write the class and method prologues, in other words, the commentary that is used in automatically creating the API documentations. After this the whole design is reviewed by the feature team and if needed, external verification by a chief architect, a domain expert and customers or users can be requested. Finally a to-do list is created for developers owning affected classes. [Palm02]

The final phase is called a build by feature. In this phase the feature team starts implementing the designed functionalities according to their personal task lists. After implementation thorough code inspections should be executed and the developers should write unit tests to gain immediate feedback. The unit test could be written before implementation, in a test first development way, or the tests could be written by another developer, in order to ensure that both of the developers agree on the design and functionality of feature. After the unit test run without failure the classes are sent to the chief programmer who is responsible of integrating the classes into the feature and promoting the feature to build. These last two steps are repeated for every feature and these may be as rapid as a 30 minutes long model storming session and a few hours of implementation before promoting the feature to build. The time taken for design sessions and preliminary modeling can vary very much depending on feature complexity and on the feature team experience. [Palm02]

The FDD uses specific milestones for accurately keeping track on the state and progress of different features. According to Brooks the developers are honest with their progress when the milestones are so sharp that they can not lie. In his opinion the milestones have to be concrete, specific, measurable events that are easy to verify [Bro75]. The recommended milestone percentage from Palmer and Felsing [Palm02] is shown in Figure 9:

| Domain walkthrough | Design | Design Inspection | Code | Code Inspection | Promote to Build |
|---|---|---|---|---|---|
| 1 % | 40 % | 3 % | 45 % | 10 % | 1 % |

**Figure 9 FDD specific feature milestones and corresponding feature readiness [Palm02].**

The completion percentage is the sum of completed milestones. So the current stage of work is not taken into account when talking about progress, even though it may be 99 % ready. The percentage given by Palmer and Felsing is of course just a starting point, the actual percentages may vary a lot depending on what kind of software and for what industry you are building. The milestone percentages can be alternated, but this should not be done during development cycle, but rather in project review meetings, iteration review or during a larger release. [Palm02]

## *2.4 Agile Modeling*

Agile modeling [Amb07a] is a methodology developed by Scott W. Ambler. Agile modeling is more like a sidekick for projects than a full grown methodology, and it has been designed to be tailored to other processes. Ambler has also introduced Agile Model-Driven Development methodology (AMDD) which is shortly introduced in Section 2.5.3.

Agile modeling borrows quite much from Extreme programming [Beck99], [Beck05]. It also is based on values, principles and practices and most of the values and principles are the same as in XP. [Amb02]

The Agile modeling values are communication, simplicity, feedback, courage, and humility. Each of these values has a simple idea behind them [Amb02]:

- The main purpose of the modeling is to communicate your thoughts effectively to stakeholders.
- The best results come from the simplest solutions. Too much complexity makes the systems hard to understand.
- The models are just abstractions of the design and are not normally correct or incorrect, only better or worse. Feedback from fellow co-workers, audience, implemented prototype or test cases will improve your models and prove them good.
- Developing the simplest thing possible, avoiding unnecessary documents, trusting co-workers' feedback and skills, admitting that you have been wrong or made a mistake are all hard tasks that require courage. Courage helps build trust among the peers and creates an efficient environment for software development.
- Humility is quite close to courage by pointing out that everyone makes mistakes and nobody knows everything. Everyone has their own areas of expertise and each is of some value for the project.

The values presented above lead to principles. Following the principles helps us to live with the values. The Agile modeling principles have been divided into eleven Core Principles and two Supplementary Principles. There are also five principles that are marked deprecated. The following table lists the core and supplementary principles with a short description [Figure 10]:

| **Core Principles** | |
|---|---|
| **Assume Simplicity** | Assume that the simplest solution is the best solution. Avoid overdesign and too complicated patterns. Keep the models as simple as possible. |

| | |
|---|---|
| **Embrace Change** | Change is evident and requirements always evolve over time because of business environment changes and stakeholder's understanding evolves changing goals or success criteria. Your approach to development must reflect this change. |
| **Enabling the Next Effort is Your Secondary Goal** | Even though the primary goal is to make working software, the secondary goal is to think about the future. The system should be robust, maintainable and documented well enough. |
| **Incremental Change** | Idea behind incremental change is that you can not do perfect models at once. There is no need to try to make very detailed and seemingly perfect models. Instead make models that are good enough for the purpose and incrementally improve it. |
| **Maximize Stakeholder ROI** | The stakeholders are investing resources on the project to get a system that suits their needs as well as possible. They have the right to decide how to use the resources and they also have a power to make the decisions on what is important in the system and what is not. |
| **Model With a Purpose** | Instead of worrying about the details of the design artifacts created, developers should think why the artifacts are needed and who gets value from them. The artifacts can be created to simplify own thinking, to provide information to upper management or to enable the maintenance team to work more efficiently with the system. If you can not think of any purpose for designing or no stakeholder who needs the artifact, the artifact is not probably needed. |
| **Multiple Models** | You usually need multiple models to model each aspect of the system. You should be able to produce many kinds of models. Also you do not need to always produce all models, but just the ones that are required for the task at hand. |
| **Quality Work** | The customer is not happy with bad work. Also the co-workers can not get help from bad design documents that need |

| | |
|---|---|
| | refactoring or are really hard to understand. |
| **Rapid Feedback** | Feedback is very important when trying to create quality software. By working together on the same model you will get instant feedback on your ideas and designs. Also working close to customer helps to understand the customer's needs better and to get feedback on the work already done. |
| **Working Software Is Your Primary Goal** | Your primary goal should be working software that meets the needs of the customer. Extensive documentation, precise status reports or flashy UML diagrams are not the primary target. All unnecessary tasks should be avoided as they are considered waste [Poppen03]. |
| **Travel Light** | Each created artifact that is decided to keep has to be maintained over the time. So the more you have models the more you have to work when a change occurs. This agility vs. convenience of having models available when needed is the balance you have to think through. |
| **Supplementary Principles** | |
| **Content is More Important Than Representation** | There is many ways to represent any model (low fidelity vs. high fidelity). Keep the representation simple enough as the most important part of the model is the content. You probably will not even need expensive CASE tools but just plain whiteboard will do. |
| **Open and Honest Communication** | The people should feel free and safe to offer suggestions and to work effectively. To build this kind of environment you need to be open to e.g. new ideas, delivery of bad news and current status. Open and honest communication enables people to make good decisions as they have more precise and correct information available. |

**Figure 10 Agile modeling principles and descriptions according to [Amb07a], [Amb02]**

The principles above can be transferred to everyday work through a set of practices. There are 13 core practices and five supplementary practices as well as four deprecated practices. The practices are quite close to Extreme programming and they usually need other practices as well to work effectively. Some of the relationships between core practices can be seen in Figure 11. The practices adopted always rely on the environment the project is working in and in cases we want to incrementally adopt few practices the practice relationships or practice clusters are useful [Els07].

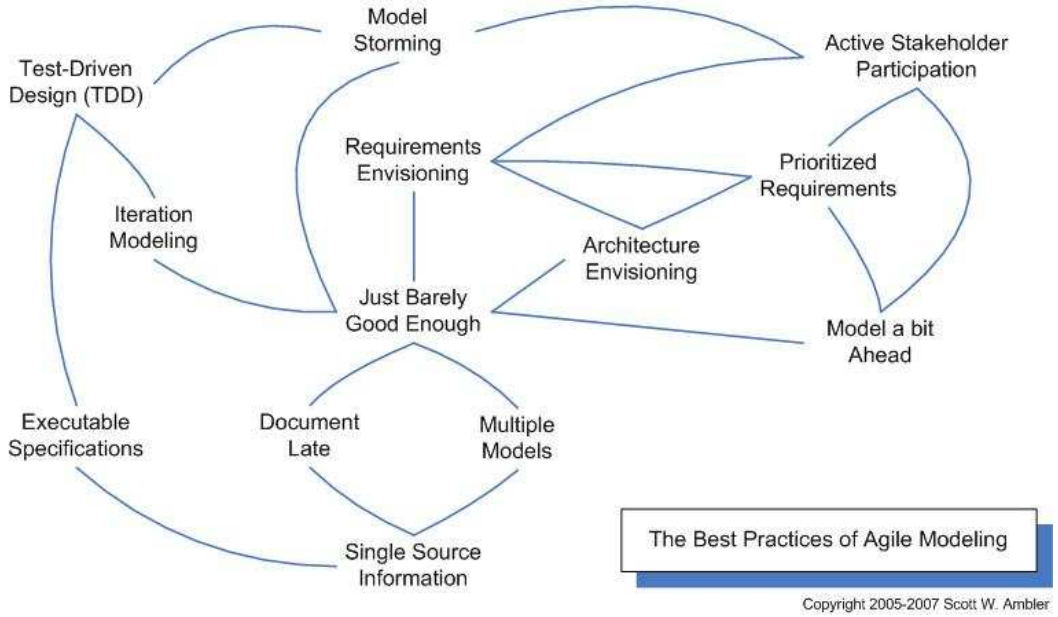


**Figure 11 Agile modeling best practices and their relationships [Amb07a].**

The core and supplementary practices of the Agile modeling are described in Figure 12.

| **Core Practices** | |
|---|---|
| **Active Stakeholder Participation** | This practice is an expansion to Extreme programming's On-Site Customer which tells to have an on-site person that has the authority and ability to make decisions about the system. The practice is expanded to have all project stakeholders (incl. users, their management, operations, senior management, support etc) actively involved in the project. |

| | |
|---|---|
| **Apply the Right Artifact(s)** | Each artifact should have its own specific purpose. In some cases source code is the best artifact to describe the functionalities, in some other cases e.g. UML deployment diagram. You should choose the correct artifacts to aid you in the task at hand. |
| **Collective Ownership** | Everyone can work on any model or any artifact on the project if needed. |
| **Create Several Models in Parallel** | No type of model is enough to represent the whole system and each of the type have its own advantages and disadvantages. It is far more productive to develop multiple models simultaneously to model the problem area than to stick with just one type of model. |
| **Create Simple Content** | Keep the content of the models as simple as possible. Do not model any additional details; unless you have to, i.e. there is a reason for modeling them. This is quite similar to Extreme programming's Simple Design practice. |
| **Depict Models Simply** | Use only the simple subset of diagrams to model the problem area. There is no need to make the models too detailed or use all the possible features e.g. UML offers. |
| **Display Models Publicly** | You should display your models publicly in a modeling wall. This is one way to support the Open and Honest Communication practice. This modeling wall should be accessible to all the project members and stakeholders. Also virtual modeling walls can be used, especially when doing distributed development. |
| **Iterate to Another Artifact** | Whenever you are stuck you should work with another artifact for a while. This change of view will let you progress and it also gives you a different view on the problem at hand. |
| **Model in Small Increments** | Agile development is by definition incremental development. Same should go with the modeling too. Model only small portions at a time, preferably the portions that you will release during current iteration or the following iteration. |

| | |
|---|---|
| **Model With Others** | Three main reasons for modeling are: modeling to understand something, modeling to communicate your thoughts or modeling to create common vision of the system. Two of these are by definition group activities where you would like to have many opinions and many people contributing to the model. You should have at least a person with you when you're modeling. This is quite close to Extreme programming's Pair Programming practice, but it doesn't limit the number of people participating. |
| **Prove it With Code** | Model is an abstraction of an idea or design. It is not usually right or wrong. To determine if the model is good, you should prove it with code. Show the result for interested to get feedback from it and improve both the model and the code implementing it. |
| **Single Source Information** | Store information only to one place. So apply right artifact to model a concept once and only once and then store the information to the best possible place. Remember the following practices: Discard Temporary Models, Create Simple Content, Create Several Models in Parallel and Apply the Right Artifact(s). |
| **Use the Simplest Tools** | Most modeling can be done in front of whiteboard without a need for expensive CASE tools. If you want to save any whiteboard model, you might use e.g. a digital camera. Whiteboard helps you to discard temporary models and is a lot easier tool for collaborative modeling than a computer. The drawing tools can be used to present important information to project stakeholders or when you need code generation. |
| **Supplementary Practices** | |
| **Apply Modeling Standards** | This practice is renamed from XP's Coding Standards. The developers should agree and follow common modeling standards. This eases the communication through models as everyone has a common modeling language. |

| | |
|---|---|
| **Apply Patterns Gently** | Design patterns are good tools for modeling and design when appropriately applied. However these should be used with care. The use of design patterns is not the aim, but the design patterns are more or less suitable solutions to common problems. In case you suspect some pattern would suit your situation, you should add the simplest part of the pattern to get the solution working and later on refactor the solution more towards the design pattern if needed. |
| **Discard Temporary Models** | Most models created when modeling are temporary models. They have already fulfilled their purpose when finished and do not add any value anymore after that. These models quickly go out of synch with the code and require updating. If the models do not add value to the project, they should be discarded. |
| **Formalize Contract Models** | Contract models are needed when some external group (e.g. another team, company etc.) controls a resource that your system requires, such as a database, another application or information service. A contract model could be e.g. an application programming interface (API) document, a text layout description, an XML schema or a database description. This contract model should be implemented using some advanced tool because it needs to be updated and maintained. |
| **Update Only When It Hurts** | The models should be updated only if it is really needed. You can live with imperfect models as long as they are good enough. Do not waste your time on meaningless updates. |

**Figure 12 Agile modeling practices according to [Amb02] and [Amb07a].**

Agile modeling is full of practices but not all practices need to be adopted. The practice adoption should be based on needs and should be incremental. It is also useful to understand the relationships between different practices in order to select suitable cluster of practices to adopt at a time [Els07].

## *2.5 Popular agile methods*

There are currently quite a large number of different agile methods used in the industry. Agile has crossed the chasm and became as a part of mainstream software development processes [Amb07b]. Most of these are based on Scrum and/or Extreme Programming or are some kind of other hybrid processes [Ver07]. This section describes the most popular methods and briefly introduces some other well known methods.

### 2.5.1 Extreme Programming

Extreme Programming (XP) was the most widely spread and well-known agile software development methodology [Amb06] however lately Scrum has become more popular than Extreme programming [Ver07]. Extreme programming was created by Kent Beck, Ward Cunningham and Ron Jeffries during a payroll software development project for Chrysler. Kent Beck became the project leader in March 1996 and began to adjust the development methodology of the project. From the methodology developed, Beck wrote Extreme Programming Explained [Beck99] that was published in 1999.

XP defines five values: Communication, Courage, Feedback, Respect and Simplicity. [Beck05] The values are a fundamental knowledge and understanding as a base of XP [Fowl05]. By starting off just with values is hard, as these can be applied in many ways. For this XP includes twenty-four practices for daily use. The practices are used to encourage and to apply basic values included in XP. The bridge between these practices and values are the fourteen principles as can be seen from Figure 13. The practices are close to traditional software development best practices, but taken to more agile level. This leads to development process more responsive and software that is better or similar quality.

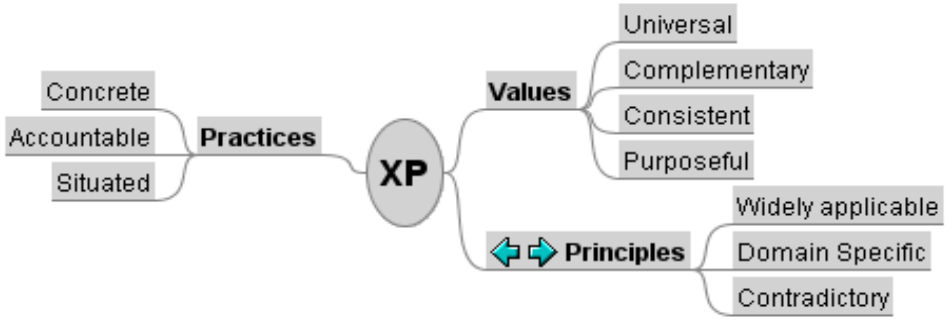


**Figure 13 The relationship and attributes of XP values, principles and practices [Beck05].**

The name of the methodology comes from the idea of taking every value and principle in the agile manifesto to the extreme level. One main value emphasized in XP is adaptability. XP treats change as a constant factor in software development projects and tries to embrace changes all the time, throughout the project. This is done by defining only a small set of small, informal requirements (stories) at a time and implementing those in small iterations. This leads to an incremental design that is affected by almost instant feedback from previous iterations. [Beck05]

Also one interesting part of XP is the Test Driven Development (TDD) that states nothing is to be coded unless there is a unit test that tests the functions to be implemented. This creates a new way of developing the software incrementally and at the same time making sure that every part of the software has been tested at least in some way. This also results in more instant feedback from the system. Test Driven Development is often connected with automated acceptance testing and called Acceptance Test Driven Development (ATDD). ATDD brings the TDD focus from a bit criticized [Ber07] low level details to higher functional level of the system. [Kos08]

XP is also criticized, because it does everything in so extreme way. The change in software development culture is huge when transitioning from traditional ways to XP. There is not enough focus on structure and needed documentation, as everything is done in just-enough way. XP requires that most developers are senior level. The more novice level developers there are involved, the more strict process is needed. The same goes

with the project size. When working in larger projects there are more processes needed and more formalized communication in order to keep project running and people aware of the situation of the project. Also there may be some problems on the business side too. XP introduces an on-site customer. The idea is that the customer sits with the team and provides feedback and clarifies requirements. This idea is usually hard to introduce to the customer as the customers are used to working in traditional way by giving requirements, paying a fixed price and then finally receiving the working software. [Step03]

Because of the extreme nature of XP, it does not fit to all purposes. XP works best with small non-critical projects with senior staff and a lot of changes. The Figure 12 below describes the suitability of XP as process for projects. The more closer the project gets to the center of the circle the more suitable XP would be for that project [Ang06]. Nowadays however, XP practices are used in many projects to add value to current development processes. Especially Scrum and XP have seen to support each other. Scrum gives a management framework where to work and XP gives day-to-day development practices [Kni07].

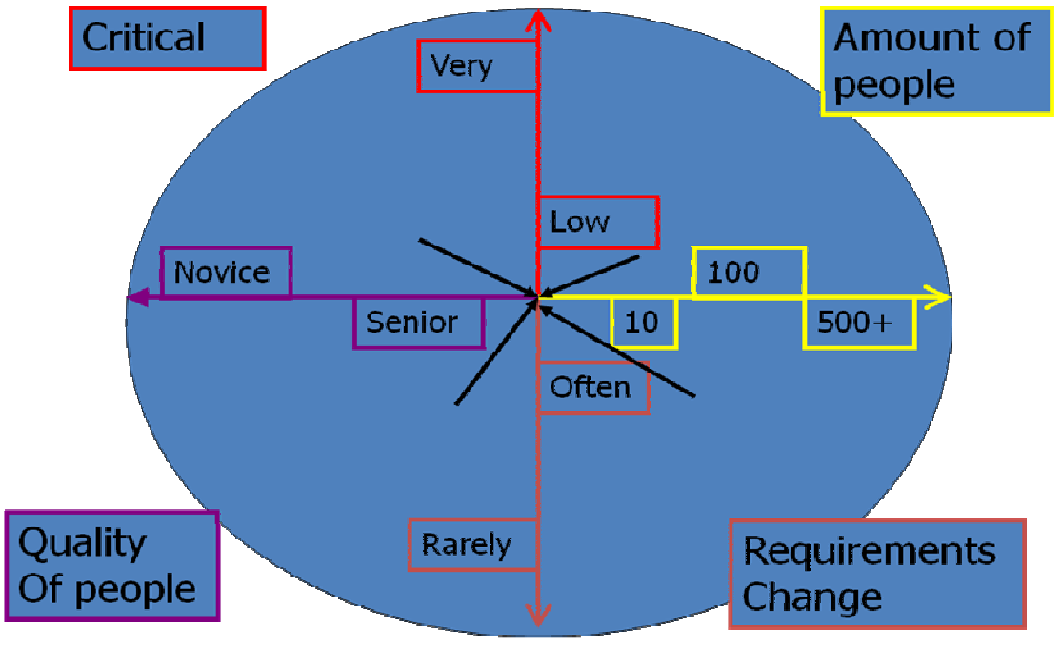


**Figure 14 XP is used in projects close to the center [Ang06].**

The process described in this thesis might use XP practices to make it more responsive. See the Section 4.2.2 for more details.

### 2.5.2 Scrum

Scrum is an agile process that is meant to be used to manage and control software development. It was first introduced as a project management style in auto and consumer product manufacturing companies by Takeuchi and Nonaka in The New New Product Development Game [Tak86] in 1986. This was refined in 1995 by the same authors in The Knowledge Creating Company [Tak95]. In same year Scrum was first formulated and presented to the Object Management Group (OMG). Later on, in 2001, Ken Schwaber and Mike Beedle described the full Scrum process. [Schw01]

Scrum is described as a “hyper-productivity technique”. Scrum tries to increase the productivity of an important product. This is done by implementing a framework that empowers teams and thrives on change. This is done by using simple techniques, e.g. small teams, daily status meetings, not interrupting peoples who are working and a single source of prioritization. [Schw01]

Scrum is mainly based on team empowerment and adaptability. The team empowerment means that once teams are assigned work, the responsibility on how to do the task is owned by the team. The team does the best it can during each increment. The only communication with the management, during the team is working, is in case there is something blocking the teams work and it needs to be removed. [Schw01] Lately the emphasis has been to describe Scrum as a framework that reveals problems in development organizations, e.g. [Dru07].

The Scrum process flow itself is fairly simple. The process starts from creation of common vision of the project. The vision describes what the customer, or the product owner in Scrum terms, wants. From this vision the product owner makes a product

backlog which is a prioritized list of high-level functional requirements. The team with the product owner selects a few items that the team believes it can implement in one sprint (iteration, length usually one month). These selected backlog items are split into tasks and the tasks are added to sprint backlog and estimated. The sprint itself consist of team working and keeping short daily Scrum meetings where the team members tell what they have done during previous day, what they will be doing this day and what impediments do they have. This is so that the whole team knows how the sprint is progressing. At the end of the sprint the team should deliver potentially shippable product with the features selected. The features are demoed in a demo session where team gets feedback from the project stakeholders. After this the team will held a retrospective session where the team discusses and reflects the last sprint and decides on improvements to the process [Schw01]. The Figures 15 and 16 present the Scrum flow.

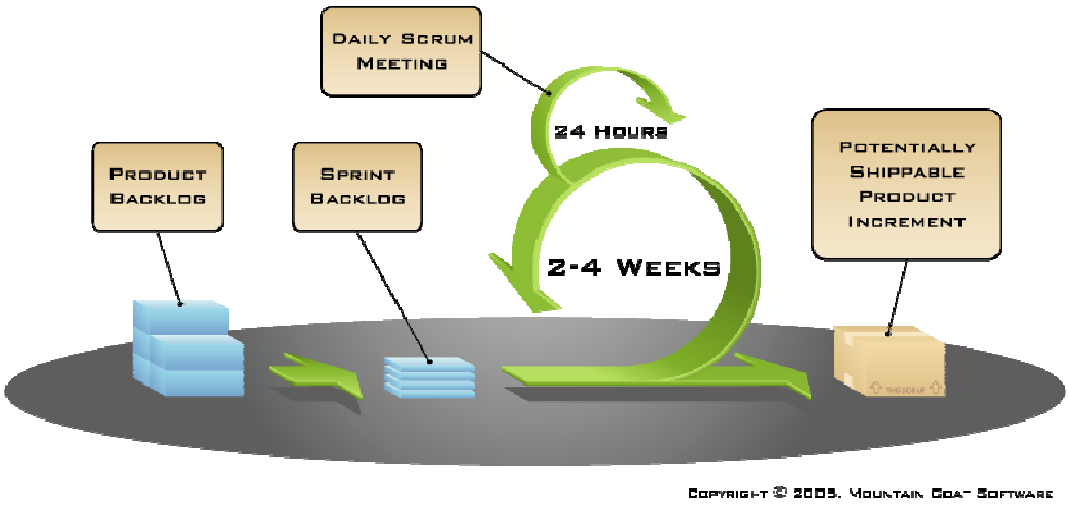


**Figure 15 Scrum flow from [Mou08].**

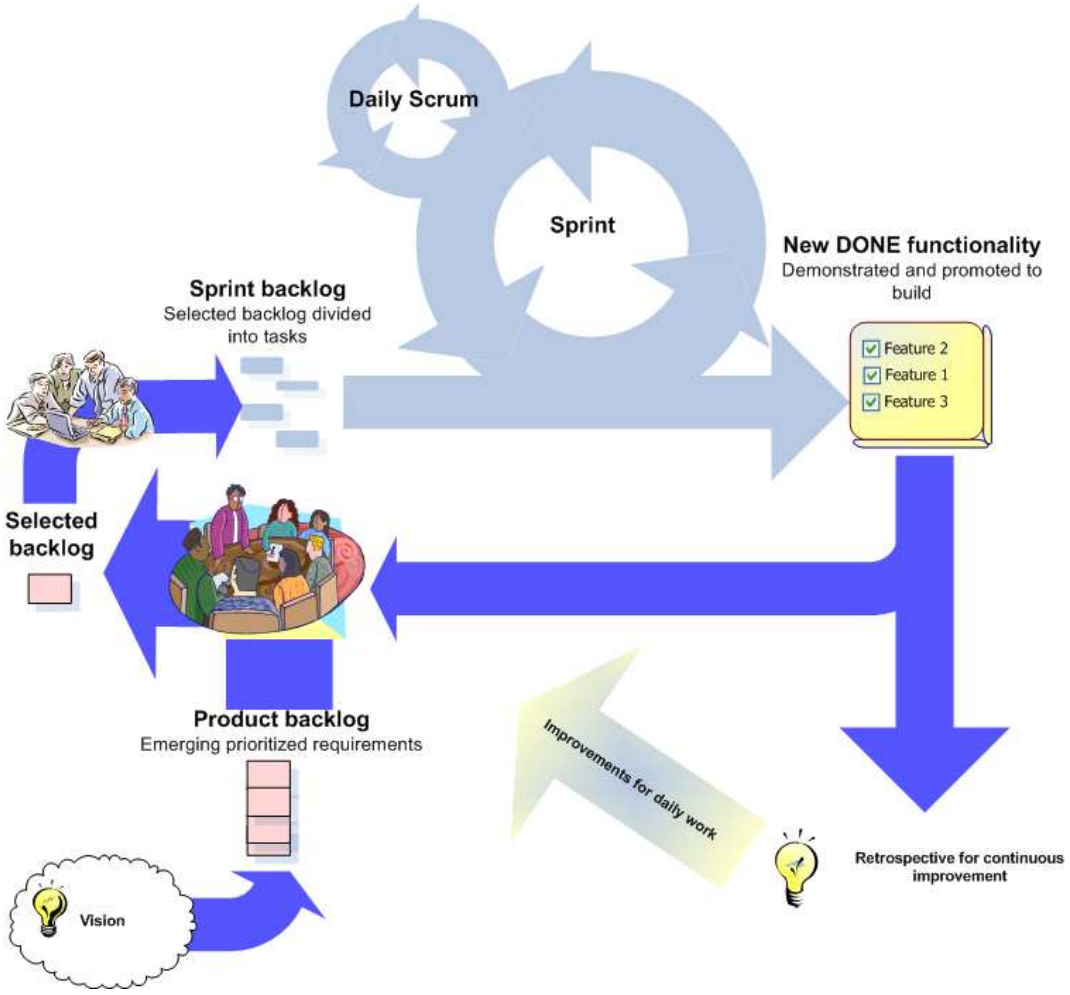


**Figure 16 Scrum flow modified from [Schw01].**

### 2.5.3 Other methodologies

This Section describes shortly different aspects of other famous agile methods.

**Adaptive Software Development**

Adaptive Software Development (ASD) is a software development process developed by James A. Highsmith [High00]. It was developed based on the rapid application development described by Highsmith and Sam Bayer [Bay94]. It emphasizes continuous adaptation of the process to the work. A normal waterfall type process is

replaced by several repeating “speculate, collaborate and learn” cycles. These dynamic cycles aim for continuous learning and adaptation to current state of the project. Highsmith describes an ASD life cycle as mission focused, feature-based, iterative, time boxed, risk-driven and change-tolerant. [Abra02]

ASD offers quite good principles and ideas, but it does not describe how to transfer them to day-to-day work. It provides an introduction on how an adaptive organization culture would be built and it can be seen as a useful resource for management in agile organizations. [Abra02]

**Agile Model Driven Development**

Agile Model Driven Development, or AMDD, is an agile version of Model Driven Development (MDD). The MDD starts with extensive modeling before writing any source code. The difference with AMDD and MDD is that in AMDD you create models that are barely good enough to drive the development forward. [Amb02], [Amb06]

The following Figure 17 describes the lifecycle of AMDD project. The cycle 0 targets to create an initial vision for the project with initial requirements and a first draft of the architecture of the system. [Amb06]

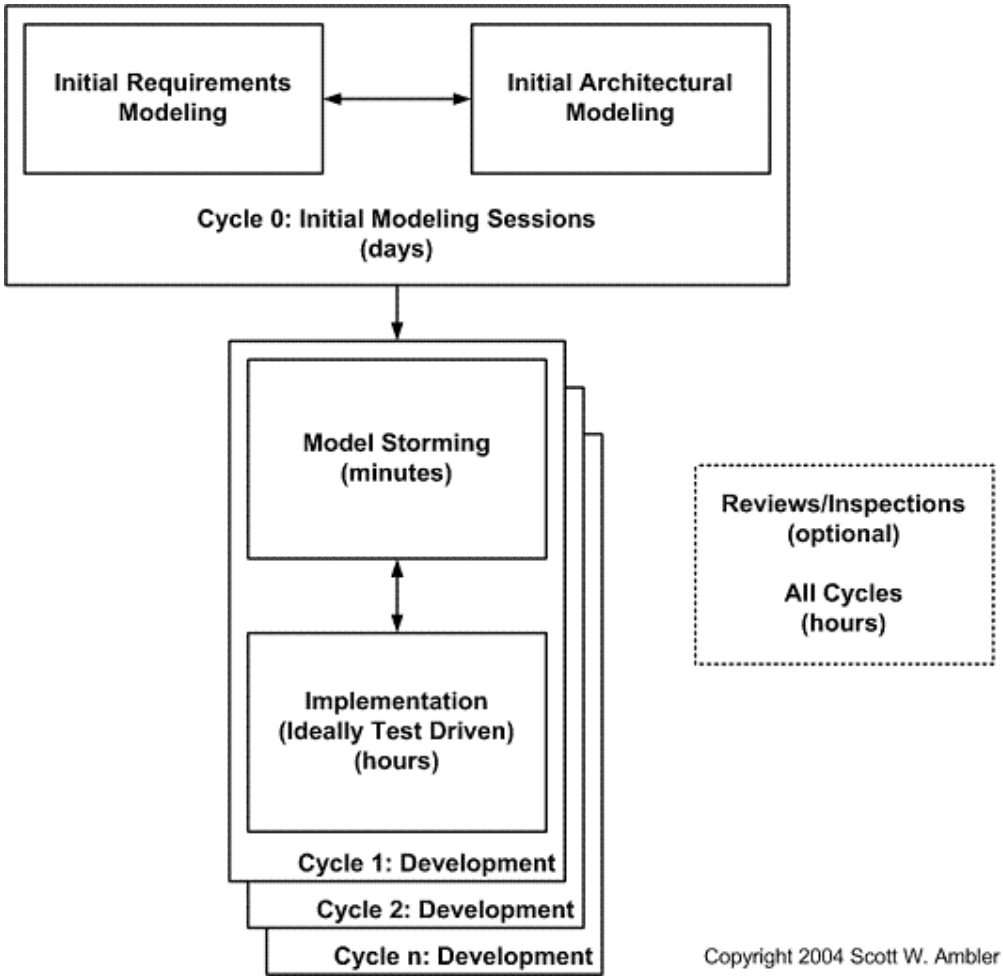


**Figure 17 The AMDD lifecycle: project viewpoint.**

After the Cycle 0, we will iterate through the rest of the project. Iteration consists of iteration planning and a model storming to support the planning. This phase requires stakeholder participation and it will lead to evolving requirements and just enough models to move forward. Implementation is done after this storming in Test-Driven Development fashion. After this the result is reviewed and verified. [Amb06]

### Agile Unified Process

The Agile Unified Process (AUP) [Amb05] is a lighter and simplified version of the Rational Unified Process (RUP) [Kru00], lately just Unified Process (UP). Its goal is to

be a simple and easy to understand way to develop software using agile techniques and concepts, while at the same time following UP. The agile techniques in AUP are familiar from other agile development approaches, including test driven development, agile model driven development, agile change management and database refactoring. [Amb05]

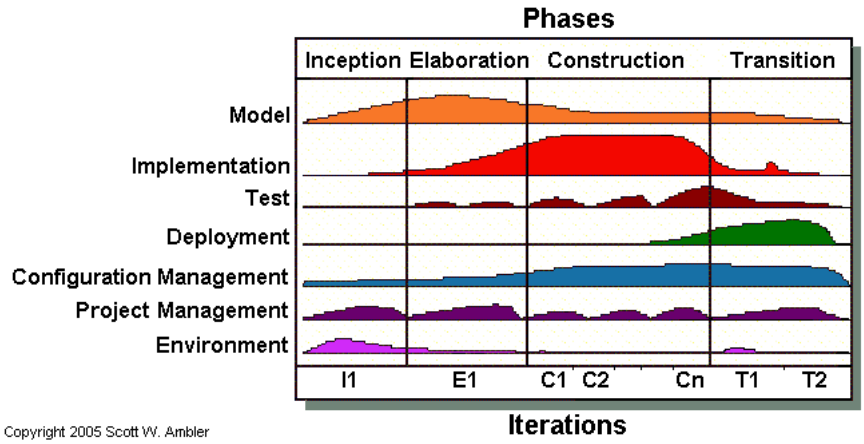


**Figure 18 The Agile Unified Process (AUP) lifecycle [Amb05].**

The AUP differs from the UP by explicitly saying that not all the documents and artifacts are needed. It has also a bit lighter ceremonies and it has changed the discipline names a bit. The idea of AUP is to make UP a bit more streamlined and agile. [Amb05]

### Crystal Clear

Crystal Family [Cock01] is a family of agile methodologies developed by Alistair Cockburn. The Crystal methodologies understand that different projects need different methods. There is no process that fits for all purposes. The Crystal Family consists of different methodologies aimed for different purposes. The family is segmented by color: The methodology for 2-6 person projects is Crystal Clear, for 6-20 person projects is Crystal Yellow, for 20-40 person projects is Crystal Orange, following Red, Magenta and Blue. Cockburn has published the descriptions of Crystal Clear and Orange. The segmentation is done according to the Figure 19 below. [Cock01]

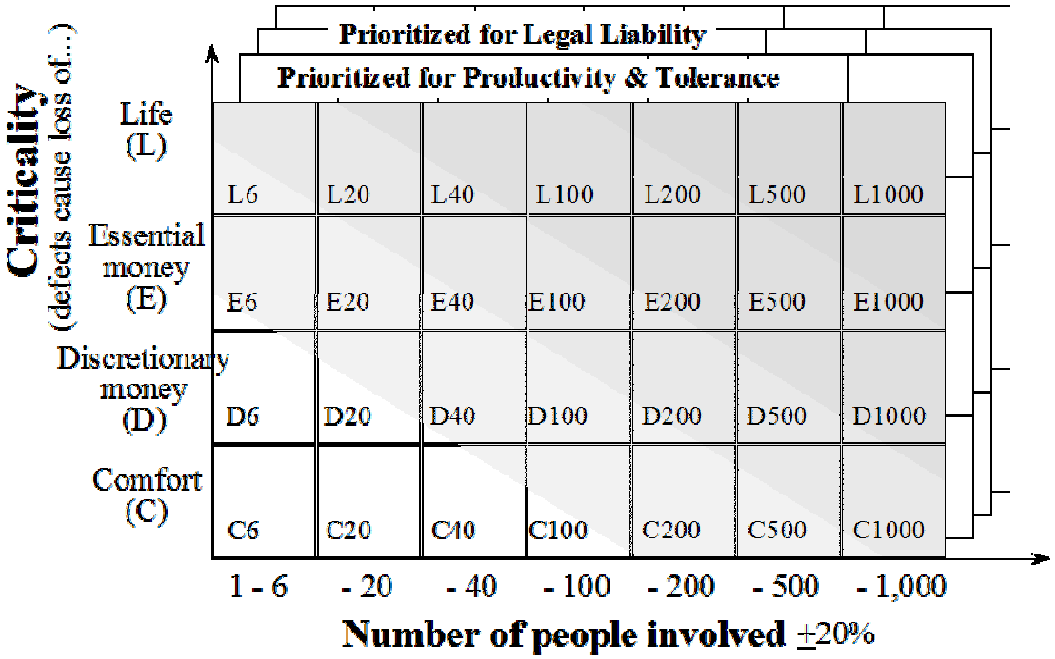


**Figure 19 Crystal family lightweight methodologies [Cock01].**

The different methods have been developed by making research on successful projects. They studied what the successful projects were dong that the unsuccessful had not done and gathered the results as a methodology. This has lead to human-centric methods for software development. [Cock01]

**Dynamic Systems Development Method**

Dynamic Systems Development Method, DSDM is a framework based originally on Rapid Application Development. It also emphasizes iterative and incremental development along with responsiveness to changing requirements as almost all agile methodologies. With this it aims to deliver system that meets the customer's needs on time and budget [DSDM03].

DSDM was developed by the DSDM Consortium by combining best-practices. The DSDM Consortium consists of vendors and experts of information systems development and it now owns and administers the DSDM framework. The first version

of the framework was published in the early 1995. The latest public version (4.2) was released in May 2003, and in 2008 an improved version called DSDM Atern was published. [DSDM03], [DSDM08] The DSDM Atern has aimed to emphasize more important values and to simplify the framework.

DSDM is divided in 3 phases: Pre-project phase, project life-cycle phase and post-project phase. DSDM has also 9 principles from which four are foundations and five starting points for the method. These principles form the bases of development using DSDM. In addition to these principles there are supporting principles, called assumptions. [DSDM03]

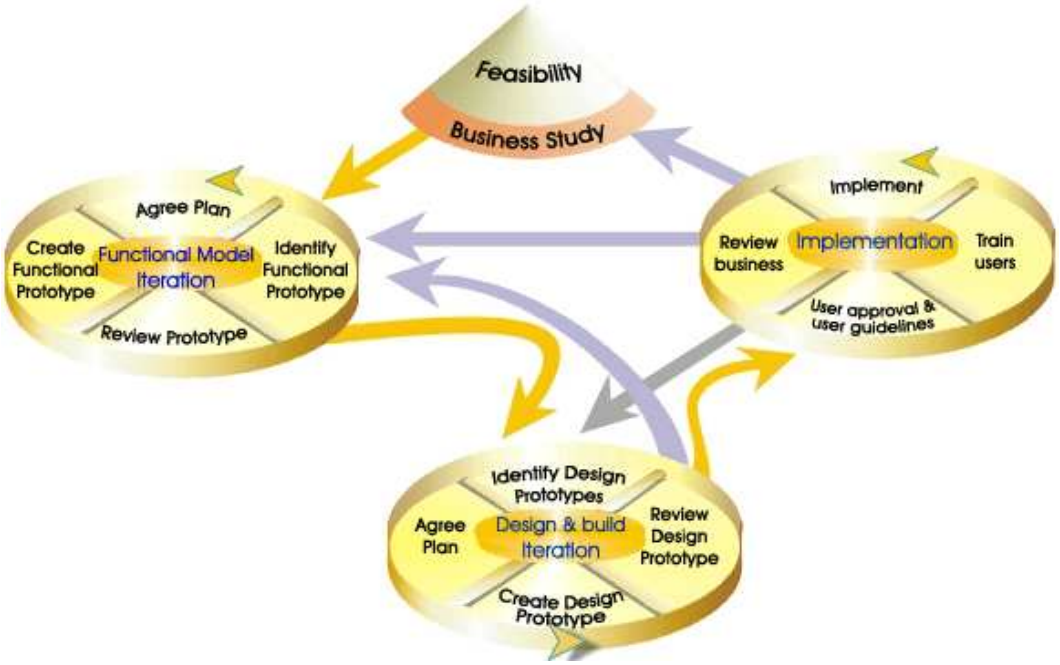


**Figure 20 DSDM project lifecycle [DSDM03].**

The pre-project phase is a short phase where basic project related issues are solved: project funding, project commitment and project candidates are identified. [DSDM03]

The project life-cycle phase is divided in five stages: The feasibility study, the business study, functional model iteration, design & build iteration and implementation stage. So the first two stages study the possibilities to make a successful project and to see how the business side of the project may turn out. The third stage is started according to

requirements identified in previous stages. An important part of the third stage is prototyping and reviewing the prototype with customers. [DSDM03]

The fourth stage integrates the functional components from the previous phase into one system. This also implements the non-functional requirements. In the fifth stage, implementation, the system is delivered and the users are trained to usage of the new system. After this stage, the project moves forward to the Post-project stage where the system is enhanced, maintained and fixed. [DSDM03]

As can be noticed from the phases, DSDM specializes to information systems, so the usage of it to pure software development can create some waste. The DSDM framework has been implemented with extreme programming, creating a highly agile development on precise process structure. [DSDM03]

**Lean Software Development**

Lean Software Development is a methodology created by Mary and Tom Poppendieck. It transfers the seven principles of lean manufacturing into software development. The seven principles of lean software development are [Poppen03]:

- Eliminate Waste
- Build Quality In
- Create Knowledge
- Defer Commitment
- Deliver Fast
- Respect People
- Optimize the Whole

These are realized in 21 useful practices that give insights on how to make software development processes more effective and lean. [Poppen03]

The Lean software development seems to gain popularity amongst agile community [Amb06], [Ver07], [Ver08], [Ver09], [Fre09] and Lean software development might be something to watch in the future as just implementing a ready method is not enough but

there is also a need to improve the way you work. This might still take a few more years as more experience papers and researches will be published. In the year 2008's State of Agile Survey by VersionOne the lean software development popularity was 1.9 % compared to 49.1 % popularity of Scrum [Ver08].

## *2.6 Pragmatic developer*

Pragmatic development and pragmatic programming are not agile methodologies or processes but rather a set of rules and advices for developers. The aim of these best practices is to encourage developers to be agile and develop software that has enough quality and offers value to clients. [Sub06]

Pragmatic programming was introduced by Andrew Hunt and Dave Thomas in 1999 [Hunt99]. These best practices were taken to a bit more agile environment and also new practices were introduced in 2006 by Venkat Subramaniam and Andrew Hunt [Sub06].

The pragmatic programming philosophy can be defined in six points [Abra02]:

- Take responsibility for what you do. Think solutions instead of excuses.
- Do not put up with bad design or coding. Fix inconsistencies as you see them, or plan them to be fixed very soon.
- Take an active role in introducing change where you feel it is necessary.
- Make software that satisfies your customer, but know when to stop.
- Constantly broaden your knowledge.
- Improve your communication skills.

"The Practices of an Agile Developer" [Sub06] book adds a point to these from agile development viewpoint: The integration should be done early and often and the project should be releasable at all times. It also adds a description on how it feels when people are agile and how to keep agility in balance and suitable for task at hand.

The pragmatic programming gives a simple and straightforward way to develop software. It can be mostly used in any kinds of organizations, whether the organization is traditional or agile. The practices are from a developer's viewpoint and they have clear benefits for developers [Sub06]. These rules will be used in this thesis to give embedded software development process a bit more developer viewpoint and day-to-day guidelines.

## *2.7 Motivation for selecting FDD as a base process*

The traditional way of software development has been proven to produce bad results in most of the projects [Sta94], [Sta04]. Agile methods have lately also proven to be more effective in producing results that suit the customer's needs [Sta04].

There has been a bit discussion if the agile methods suit the needs of embedded development, e.g. Ian Sommerville stated that agile software development methods are not suitable in large-scale system development, distributed development nor development where there are complex interactions between other hardware or software systems [Somm04]. However there have been many successful results from agile software development on embedded devices. From a Finnish point of view, one of the most interesting is the agile adoption in Nokia Siemens Networks that has been so far quite successful [Vil08] and according to surveys in the year 2006 almost 70% of the personnel in agile projects would not want to change back to old way of working [Haa07]. Also interesting results have been received from Nancy Van Schooenderwoert who studied adoption of Extreme programming in embedded development already in the year 1999 [Scho04].

Even though the previous cases have adopted Scrum and Extreme programming as their agile method, it is likely that FDD has good features to offer to embedded software development. First of all FDD is scalable so it will fit to even larger projects [Palm02]. Scrum on the other hand does not offer day to day practices for development, but it more or less stays as a high level management framework [Schw01]. The FDD offers

process oriented clear flow that can be followed easily and that has detailed instructions on how to do the work too [Palm02].

FDD can be used to address concerns about the testability and hardware interfaces by thinking these already in the initial concept modeling phase. In that phase we are able to already identify key hardware interaction points and can define abstract interfaces to hide those points. This makes the system testable also on the development host system by using mock, stub or fake devices [Kos08].

Finally in my personal experiences the embedded development projects are quite traditional projects using a very controlled process. The FDD has a quite strict and formal process but at the same time it emphasizes working software and effective communication in an agile manner. It might be a good stepping stone to more agile development process, but it does not require as much effort to adopt as many other agile methods.

Because of the previous reasons the FDD has been chosen to be basis for the proposed process in this thesis. The process is enhanced with different practices from Agile modeling, Extreme programming and Scrum as well as day to day guidelines from Pragmatic programming. It also adds a new perspective on testing in FDD projects that has not been discussed very much in FDD community.

# 3 Embedded and real-time systems

This chapter describes what are the embedded and real-time systems and where can we find them. We introduce normal constraints or design metrics of embedded and real-time systems and discuss why these are important.

Finally we introduce seven main issues of embedded and real-time software development and break these larger issues into few more or less troubling concrete problems.

## *3.1 What are embedded and real-time systems?*

An embedded system is system where a processor is completely encapsulated by the device. Usually the embedded systems are special-purpose systems that perform pre-defined tasks with very specific requirements. Embedded systems vary from ATMs to mobile phones and from tiny MP3-players to nuclear reactor security systems. More examples can be found from the text box on the right. The embedded systems vary by criticality, size, manufacturing volume, power consumption needs, CPU power, complexity of software, etc. [Barr07]

**Examples of embedded systems:**

Air conditioning systems
Anti-lock brakes
ATMs
Avionic systems
Digital camera
Electronic toys/games
Home security systems
Medical devices
Mobile phones
Photocopiers
Smart ovens/dishwashers
Stereo systems
Televisions

Even though the embedded systems vary a lot from each other, they have common grounds too. The fact that embedded systems are designed for specific purpose with pre-defined tasks makes it possible for developers to optimize the hardware and software of system in many ways. The solution could run mainly

on hardware, on special-purpose processors with minimal amount of software or it can run on general-purpose processor (GPP) with most of the work done on software side. Usually the hardware is specialized for just certain product of product family. This leads often to a situation where software is developed for hardware that is not yet available. [Yag03]

When speaking of embedded systems the term real-time system is often used alongside. This is because many embedded systems are reactive. They wait in idle mode for commands from user interface or some other events to occur and act according to those. This may need many concurrent processes to run on a single processor. [Barr07]

In real-time systems a big issue is the timing constraints. The timing constraints may be hard, soft or firm. The hard timing constraints mean that if a system does not finish a task before deadline, there will be a failure. The soft timing constraints mean that the deadlines may be violated to some degree. The firm timing constraints are a combination of previous ones resulting in that some tasks may have hard constraints and some less important soft constraints. The timing constraints lead designers to scheduling and performance issues. However, nowadays many embedded systems rely on a 3rd party real-time operating system (e.g. Nucleus RTOS, VxWorks RTOS, vrtx etc.). This leads to situations where more up front design has to be done and the designers may need more time to get comfortable with the operating system used. [Awad96]

## *3.2 Constraints in embedded and real-time system design*

Embedded systems have four common design challenges [Awad96]: Common design metrics, time-to-market, Non-Recurring Engineering (NRE) & unit costs and platform metrics. The Common design metrics are the non-functional constraints that drive the design of embedded systems. The problem is that most of these are contradictory. Most important parts of these contradictory metrics are NRE vs. unit costs and size vs. power consumption [Awad96]. The design metrics have been listed in Figure 21:

| | |
|---|---|
| **NRE costs** | The cost of designing the system |
| **Unit cost** | The manufacturing cost of the system |
| **Size** | Physical space required by the device, often measured in bytes for software and gates or transistor for hardware |
| **Performance** | Execution speed of the system |
| **Power** | Amount of power consumed. Measured in lifetime of the battery or cooling requirements |
| **Flexibility** | The ability to change functionality without large NRE costs. |
| **Time-to-prototype** | Time to build working version of the system |
| **Time-to-market** | Time to develop system so that it can be released and sold to customer |
| **Maintainability** | Ability to modify system after its release |
| **Correctness** | Is the functionality implemented correctly |
| **Safety** | The probability that the system won't cause harm |

**Figure 21 Table of common design metrics [Awad96].**

The easiest way to minimize the NRE costs would be to use a powerful general-purpose processor and do everything with simple ineffective software, but this might make the unit cost higher, increase the size and most of all increase the power consumption a lot. The same goes with any other design metric. This forces the embedded system designers to find an average combination that is as good as possible. [Yag03]

For many high volume products like mp3-players or mobile phones, the minimizing cost is the primary design goal. Every component is selected to be just good enough for necessary operations. In some products this component selection is a bit easier, as some tend to act like data-processing pipelines. For example, set-top boxes just process the transmission data received by pushing it through a series of custom circuits. Such architecture can be seen on Figure 22. When the task is to design prototype or very low volume products, then normal personal computer hardware can also be used in embedded systems. [Yag03]

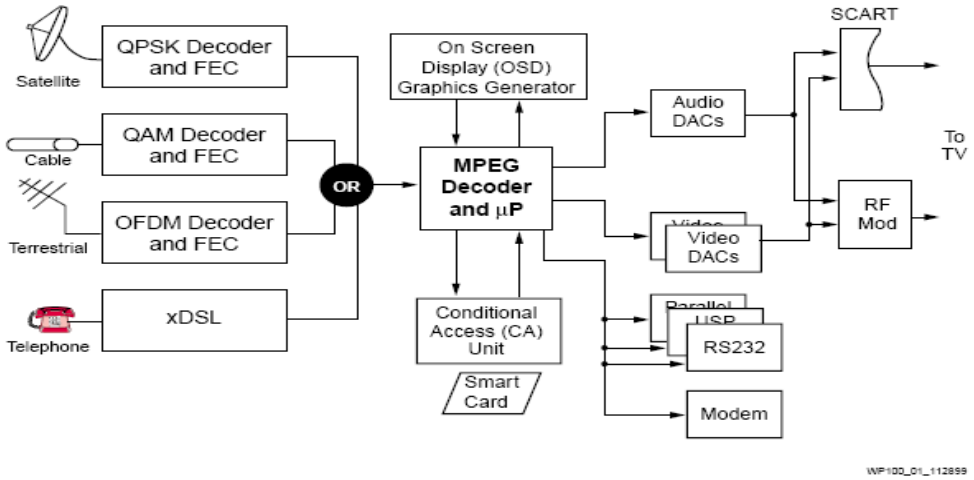


**Figure 22 Example of current set-top box architecture [Xil00].**

Another special feature in embedded systems is the lifetime of the product. Some embedded products are expected to function for just a few years, e.g. small consumer electronic devices like mobile phones, stop watches. Other products have the lifetime expectation of decades, e.g. nuclear power plant control systems, air conditioning systems and space station control systems. When developing systems with long lifetime you have to take into account the issues of software updates and reliability. [Barr07]

The challenges grow when we move from embedded systems to real-time systems. The real-time systems have operations that have deadlines and if the system does not meet the deadlines the consequences might be catastrophic. For example, if the anti-lock brakes in a car do not react in timely manner, the driver might end up in a car crash. In case the consequences of missing a deadline are not catastrophic, at least the result of the calculation is incorrect.

Real-time systems add more complex design issues to be decided in advance. One essential choice is the scheduling algorithm used. The system's timing and the worst-case scenarios can be calculated using RMA (Rate-monotonic analysis) and timing diagrams. Based on this some decisions can be made on how the scheduling is implemented: using e.g. Round robin algorithm or by using a ready real-time operating system (RTOS) and suitable mode for that. [Sim99]

Embedded and especially real-time systems are quite often implemented using a hardware platform specialized for just that product or using a different platform than previous project. Often it is needed for the developers, especially ones implementing more of the device driver level code, to study the performance, behavior and use of the platform. [Barr07]

From real-time perspective the interesting features of the platform and selected operating system are worst-case performance, interrupt latency and context switch time. Worst-case performance means the worst-case time from the moment when an event happens to the moment that the system has responded to the event. Interrupt latency means the time operating system uses to process a new arrived interrupt. Context switch is the operation that happens when a process is suspended and a new process is given some run time and of course the time it takes is highly important in embedded systems. [Barr07]

## *3.3 Issues in embedded and real-time software development*

The design and development of embedded and real-time systems has been seen as a problematic area. Jerry Krasner from Embedded Market Forecasters has studied embedded software projects and has found out that 13.1 % of the projects are cancelled and 54.0 % are completed behind schedule. The delay average is 3.9 months. As Krasner stated in his final words of the report: "It is clear from information provided herein that embedded software practices, being much less methodological than hardware design processes, are responsible for design delays and missed 'windows of opportunity'." [Kra03]

What are the main issues in the embedded and real-time software development? I've identified 7 main issues affecting the failure of the embedded and real-time software projects:

1. Response time and timing related issues

2. Platform architecture related issues
3. Embedded software development tools related issues
4. Development process related issues
5. Programming and design practices related issues
6. Project size and complexity related issues
7. Issues from typical constraints

The 7 main issues consist of many different parts. The Figure 23 shows the big picture and the following Sections describe the issues in more detail. The list has been collected from multiple sources: [Barr07], [Osh06], [Sim99], [Som04], [Stew99a] and [Stew99b]


**Figure 23 Issues in embedded and real-time software development.**

### 3.3.1 Response time and timing related issues

Response time and timing related issues refer directly to the real-time related issues, but the same issues can also affect typical embedded projects. The response time is highly dependent on hardware and software design decisions. The designers must find suitable software and hardware architecture; select the correct components; select or implement correct scheduling, and make sure that the buses enable fast enough communication. [Osh06]

The main reason that make the response time an issue is that there often is no final hardware available. The developer can not make execution time measurements and can not really test the timings. In fact, it is often a problem even the hardware or at least a prototype would be available. Developers might not have any measured execution times just estimates. Also the measurements are left to the end where all the features have been implemented. However at this stage there usually are so much timing problems, that it is really hard to solve them. [Stew99b]

Another problem affecting response time is that the hardware platform specialties haven't been analyzed. Platforms differ quite much on how fast they do certain calculations with different kind of numbers. Knowing what is fast and what is not is very important when implementing real-time software. Using correct and fast instructions in calculations saves a lot of performance tuning. However, it is important to remember that fine-tuning and optimization should be done only when a bottleneck is found. [Stew99a]

The hardware specialties also affect the memory usage. There might be significant performance differences in accessing ROM or internal RAM memory and even more when accessing external RAM. Memory analysis aids in using the memory as effectively as possible. If the platform offers a cache, the memory analysis is even more important part of the design and if not made properly will cause some performance issues. [Stew99b]

### 3.3.2 Platform architecture related issues

Even more embedded and real-time systems rely on multiprocessor platforms which mean that the application might be partitioned on multiple processors. This leads to multiple design challenges for designers. The communication between multiple processors has to be well defined and consistent. Normal issues arising from this are byte endianess problems, byte ordering and padding problems. Also the partitioning must be done with care to avoid bottlenecks and to ensure even load distribution between processors. The final issue with multiple processors is that there is still only a common set of hardware resources available. The hardware resources have to allocated and managed so that it enables maximum utilization without any problems. [Osh06]

The modern platforms and the modern business environment bring us also more issues. Often the platforms change from project to another and include quite much custom hardware. It is hard to find support for the platform, as it seems like it is unique. Especially, when the development team is small and there are no other low-level software specialists around. Also the different environments and constant change affecting the environments leads often to generalization of the software. The aim is to enable the software to run on various different platforms. However doing this based on one project and one platform often fails and leads to hard to maintain software that is not really portable as well as lots of unnecessary work. Better choice would be to do the generalizations when there is a need for that. [Stew99a]

### 3.3.3 Embedded software development tools related issues

Desktop and enterprise software development tools are often full of features that have high level of automation and good integration with e.g. source control and server software. In embedded and real-time software development the tools are usually lacking many of those features. The choice of development tools is often made from business perspective on basis of marketing, feature promises and user base. The tool choice in embedded projects should be done based on technical evaluation. [Stew99a]

From the poor tool base comes multiple problems and one of them affects quite much the success of projects. Automated testing is often lacking from embedded software projects. Testing is done interactively using manual test cases. This is much more error prone than automated testing and causes the testing to be started at too late. [Ste99a]

### 3.3.4 Development process related issues

Development process might affect the result of a project surprisingly much. Lately agile processes have been gaining popularity from traditional processes [Ver07], [Ver08], [Ver09], [Amb06], [Amb07c], [Amb08], [Met08], and [Shi03]. There are many process related issues recognized in embedded and real-time projects that have failed. Few are described in the following paragraphs.

Code reviews and inspections are usually an important part of the process if done correctly [Poppen07]. They can find common vulnerabilities, especially race conditions, buffer overflows and memory leaks in embedded and real-time development. Also static code review tools can be used to perform code reviews and to find possible problems. The code reviews have been noticed to improve productivity and quality. For example IBM has found that inspections gave a 23 % increase in productivity and 38 % reduction in bugs detected after unit testing [Gan01]. Still the code reviews and inspections are often forgotten and left aside when deadline is approaching. Developers have a tendency to protect and hide their code and at the same time the code robustness and correctness suffers. [Stew99b]

Documentation is a common issue in many projects. Often it is said that there is too much documentation and sometimes there is too little documentation. The main problem is that the documentation is written after implementation instead of writing it during and before the implementation [Stew99b].

A normal problem with traditional projects is the constant sense of urgency and high number of overtime hours. These hours usually are not as productive as the normal hours and can cause the developed software to be of lower quality [Beck06]. In embedded software development it can take a long time to solve some hardware related problem. Often developers think that they can not take a break while the problem is not solved [Stew99a]. Even though there might be a deadline approaching a short break might let you take a needed distance to the problem at hand and help you find an alternative solution to the problem [Stew99a]. I would compare a software project to a marathon: You have to have a sustainable pace and it will not hurt much if you stop for a while and have a drink. In fact it will hurt you more if you run as fast as possible and do not stop for a drink of water.

Task switching has been identified as one big waste in software development by Mary and Tom Poppendieck [Poppen07]. The task switching occurs when the developers are working on a multiple projects at a time. The problem is also very common in embedded software development. According to the 2008 Embedded Systems Market study from Tech Insight/Embedded Systems Design report [Emb08] about 65 % of the respondents work on two or more projects and only 33 % were working on single project. More than every tenth (13 %) work on three or more projects. Working on many projects simultaneously causes task switches to occur and is considered a wasteful activity in software development [Poppen07].

### 3.3.5 Programming and design practices related issues

The development process gives us guidelines how the project is implemented. The programming and design practices give us guidelines on daily work [Beck05]. The practices guide the day-to-day work of anyone involved in project. Multiple problems arise also from incorrect working habits.

The common design practice related problems are identified by David B. Stewart in his two part article "30 Pitfalls for Real-Time Software Development" [Stew99a, Stew99b]:

- Only one design document
- Dependencies
- One solution applied every time and the first solution is right
- Overdesign
- Reusing non-reusable modules

One document type is quite important for the project success, design documents. Many software systems are written based on one or none design diagrams. This approach does not give the benefits of modeling and system design to the developers by giving only one view to the system [Stew99b]. In addition this single diagram mixes multiple diagram types and is not easy to understand. A simple syntax or a legend box could help others to decipher what the diagram means [Stew99b].

Dependencies are almost inevitable in software. The dependencies limit the use of modules and must be taken account when talking about reuse of modules. Understanding the dependencies of software also help to find out how to test modules separately and how the error handling should be implemented throughout the system. However, if there are too much circular dependencies, the dependencies cause the modules to be hard to test and hard to reuse. [Stew99a]

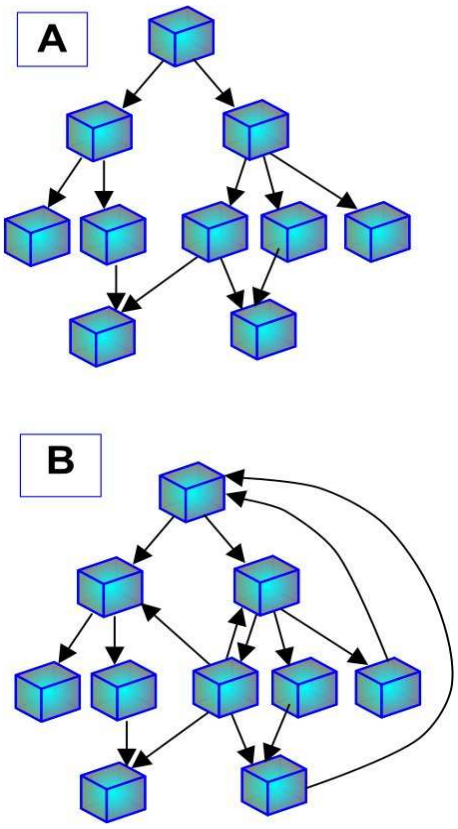


**Figure 24 Dependencies**
**In case A are normal, no circular dependencies. In case B there are many caircular dependencies leading to complex design and reusability problems.**

After finding a solution to a problem, it is normal to try to use the same solution in similar situations. When using e.g. design patterns this can lead to overdesign when trying to use same solution to every problem [Beck05]. In embedded world it might take a long time to get hardware interface work correctly. After the interface works,

the developer usually is happy and thinks that the task is done. However the solution rarely is the best one. It can often cause problems with high processor usage or timing problems with too high priority [Stew99b]. So to get the software right, multiple different designs should be tried to learn the problem and to implement as good solution as possible.

Overdesign is a common problem in software development, not in embedded systems general. A lot of overdesign complicates the system and makes maintenance and change tedious tasks [Beck05]. In embedded and real-time systems we want the software to utilize the hardware as much as possible with as cheap hardware as possible. So the correct solution will always be the simplest and the fastest to execute. So overdesign causes raise in hardware costs by requiring needless processing power and additional memory [Stew99a].

Not all code is designed to be reused. If some code that is badly designed or purposefully designed to be non-reusable is reused it may cause the functionality of the code to change or some missing part might cause problems. It is often more suitable solution to rewrite the module after analyzing the old module. [Stew99a]

There are quite many additional bad habits especially the embedded and real-time software developers have. Quite many of these relate to the C language as it is most common programming language for embedded systems [CMP01], [CMP02], [CMP04], [CMP05], and [CMP06]. The following problems were identified by David B. Stewart in his two part article "30 Pitfalls for Real-Time Software Development" [Stew99a, Stew99b]:

- Large if-then-else and case statements
- No style and naming conventions
- Delays implemented as empty loops
- One big loop
- Global variables
- Workarounds
- Configuration information in #define statements

- Including #global.h
- Wrong use of interrupts

Large if-then-else and case statements make the code hard to read and debug. From the real-time perspective the differences between best-case and worst-case scenarios are too large and might lead to possible timing errors or under utilization of processor. It is also harder to test the code so that the cases test whole code as the number of different branches is so large. [Stew99a]

When implementing the system the developer is responsible of the quality of the code. There are quite common naming and style conventions used widely. However not everyone has heard about them. When everyone in the team programs in his/her own way the software will eventually be so hard to test and the code will be extremely hard to understand. So to enable success a common naming and programming style conventions should be determined. [Stew99b]

Delays are used in code to ensure that another task is completed before moving forward. Using empty loops as delays affects the code portability. The same software that worked fine on some platform might have hard to find timing problems on some other platform. Better solution would be to implement the delay using real-time operating system services. [Stew99a]

One big loop is a traditional architecture for simple embedded software. The problem comes when we want to modify the execution time of some individual part, or parts, of the code. This problem arises when the processor is overloaded and there is a need to slow down the less critical parts to give time for the critical parts. One big loop does not allow this, but runs everything at the same rate each loop. [Stew99a]

Global variables are traditionally looked to violent the object oriented design models and encapsulation [Amb02]. In real-time systems with multiple processes running on real-time operating system all global variables are usually shared amongst all processes and can cause strange problems. Some developers use these variables as shared memory

however all locks, semaphores and other means to solve race conditions are something that should be avoided in real-time software. [Stew99b]

Workarounds is a common problem in software development projects. Problems are solved with a quick patch instead of searching the root cause for the problem and making the needed refactorings [Sub06]. The problem with workarounds is that the root cause will surface every now and then to cause additional problems. The workarounds should be only temporary answers to problems not permanent solutions [Stew99a].

A common practice in embedded development and in general in C development is to use precompiler statements to configure software. Using these #define statements is a bad idea in embedded world for two reasons. First one is that in case the software needs fixing as some configuration value is e.g. to small the whole software needs to be recompiled and reinstalled. As the software is meant for embedded device, this might prove to be very hard. Second reason is that the usage of #define statements and constants is very bad for reusability and porting to different platforms. [Stew99b]

A common problem is also using a global project wide header file, usually called globals.h or project.h. Such a file is included into each source code file. The practice is followed because it seems to be easy and simple, but in fact it makes the reuse of software a lot harder and takes a lot more time to maintain. [Stew99b]

The last programming related issues are wrong use of interrupts. According to David B. Stewart [Stew99b] 80-90 % of the program code is often implemented in interrupt handlers. The interrupts contain complete I/O handling and also loops. However large interrupt handlers can cause several problems like priority inversion, scheduling problems and excessive use of global variables [Stew99b, Barr07]. The handlers are also very hard to debug as many debuggers restrict the breakpoint setting inside interrupt code [Stew99b].

### 3.3.6 Project size and complexity related issues

The projects itself cause problems that are not necessarily related to process used, ways of working or directly to any technical issue. These are also quite traditional problems that also the normal software development projects struggle with. One solution helping to get through these is use of more advanced software development tools. However there lies another problem from the embedded software point of view, see Section 3.3.3 Embedded software development tools related issues.

Tom DeMarco and Timothy Lister noticed already at the late 70s that larger software projects, meaning ones that take over 25 work-years to finish, almost every fourth of them failed or was cancelled [DeM87]. According to TechInsight's Embedded Market Study 2008 [Emb08] the average time for an embedded project in 2008 was 13.1 months (12.6 in 2007) and the average lateness was 4.4 months. The design teams average is 15.2 persons (13.6 in 2007) growing also a bit from previous year. The size of the project and the project teams causes communication issues and disturbs the development.

The complexity of hardware/software systems with real-time constraints is an issue itself. As embedded systems vary from highly sophisticated mobile phones to nuclear plant control systems with tight security and response time requirements, it is clear that the complexity of the system varies a lot.

### 3.3.7 Issues from typical constraints

The typical embedded and real-time system constraints also cause issues in development projects. The common design metrics that have to be taken into account are (presented also in the Section 3.2 Constraints in embedded and real-time systems):

- NRE costs
- Unit cost
- Size

- Performance
- Power
- Flexibility
- Time-to-prototype
- Time-to-market
- Maintainability
- Correctness
- Safety

The following issues are identified to be related to these design metrics.

Error handling and detection is a crucial part of the embedded and real-time systems as the devices might be used in multiple environments and the device might be unable to inform users about the error conditions [Sim99]. There seems to be two dominating ways to implement this [Stew99b]: The developer implements error detection and handling everywhere, many times even when it is not necessary. This causes problems with the performance and timing. Another way is not to implement error detection and handling code unless needed as a workaround for issues arising during testing. This causes defects to be found from the complete product as testing can never be so comprehensive that all defects are detected. [Stew99b]

Battery lifetime and processing power require quite much work from the developer, especially when talking about systems that are mass-produced and use very low cost hardware. The software should use the processor power as effectively as possible. At the same time the software should not use the hardware for unnecessary operations and should be able to turn off some parts of the hardware platform temporarily in order to save battery time. Cutting off everything unnecessary from the software causes lots of design challenges both for the hardware and software designers. [Ten03]

Another issue related to typical constraints of the embedded systems is that e.g. consumer embedded systems can be used in multiple different environments. The testers of the device and the developers can not predict all uses of the system and so the system might behave strangely in different situations. [Koop96]

Related to the previous issue is that there might be an easy way to fix defects found after product has been released to markets. However the problem is that most embedded systems can not update itself over the air. Updating the device might require opening the casing and changing the memory circuits permanently containing the software. This kind of operation can be very costly in case the defect is major one. So the correctness of the software is a large issue in embedded and real-time software development. How to ensure that the software works correctly in all situations? How in case of a fault the system can recover and continue working? What are the reliability needs for the product? This is a great design challenge for the system developers. [Barr07]

# 4 Process for embedded and real-time software development

This chapter introduces a new methodology for embedded and real-time software development. This process is based mainly on Feature Driven Development with practices and parts from Agile Modeling, Extreme Programming and Scrum. Section 4.1 proposes a process for embedded and real-time software development.

Section 4.2 gives a detailed view of the activities done in each subprocess. Next Section 4.3 will introduce the agile practice used in day to day work in order to make the process agile.

Section 4.4 “How the process meets the development issues?” considers the issues introduced in previous Section 3.3 and how these issues are addressed in the proposed process.

## *4.1 Proposed process for embedded software development*

The proposed process is described in detail in following Sections 4.2 and 4.3. The process is a customization of Feature Driven Development [Coad99], [Palm02] with some tailored Agile Modeling [Amb02], [Amb04], XP [Beck99], [Beck05] and Scrum [Schw01], [Schw06] practices. The variations of original FDD process and the purposes for them have been described after each subprocess. The aim is to come up solutions to normal problems in embedded software development introduced in Chapter 3.

The subprocesses are presented with Entry – Tasks – Verification – eXit (ETVX) template used also in the description of FDD processes. A more detailed description of ETVX template is in Figure 25. The overall view of the process and its subprocesses was introduced in Section 2.3.

The tasks have a header and a short description what is to be done. The header contains information on the title of the task, the person responsible of the task and a section telling if the task is required or optional.

| Section | Description |
|---|---|
| Entry | Gives a description on process and a set of requirements needed to accomplish before starting process |
| Tasks | Task to be performed during process |
| Verification | Describes how to verify that the tasks are correctly done |
| eXit | Describes what deliverables the process delivers |

**Figure 25 ETVX template.**

The proposed process is described below in Figure 26 as a high level presentation. It shows the main subprocesses:

- Develop an overall model
- Build feature list
- Plan by feature
- Design by feature
- Build by feature
- Feature retrospective
- Milestone retrospective.

It also shows the basic flow of the process with main outputs from each different subprocess.

The process described in Figure 26 is basic FDD process with two new subprocesses: Feature Retrospective and Milestone Retrospective. More detailed description of all the subprocesses is in Section 4.2.

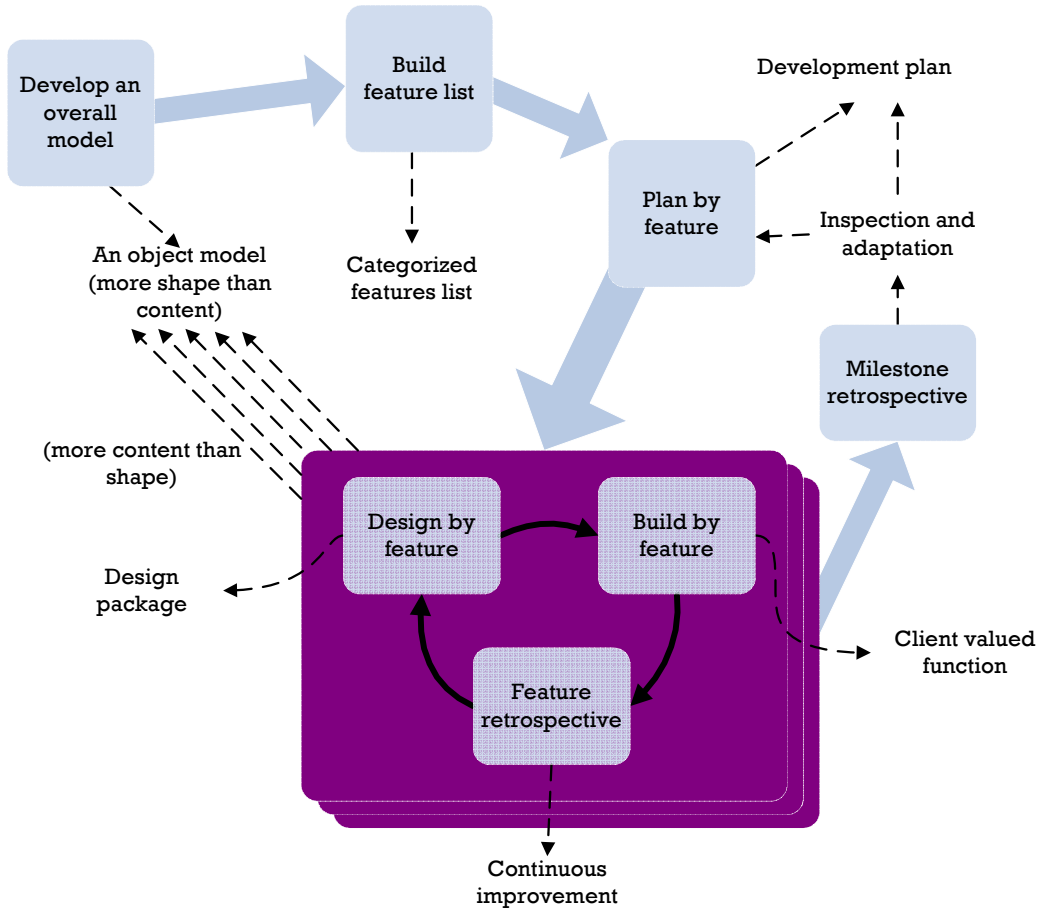


**Figure 26 Process flow of the proposed process.**

## *4.2 Process description*

This Section describes all the subprocesses of the process in detail.

### 4.2.1 Subprocess 1: Develop an Overall Model

**Entry:**

An initial project-wide phase that results in domain object model of the system. This is lead by Chief Architect and it involves the developers, specialists and Domain Experts in the design and modeling of the system domains.

The Domain Experts give a walkthrough on the whole system and its context. Then they perform a more detailed walkthrough on all areas of their domain to be modeled. According to these, the developers and domain experts create domain models in small groups. One model or a hybrid of many models is selected and added to the overall domain object model.

**Before Entry:** The roles for the project have been decided; at least Domain Experts, Chief Programmers and the Chief Architect have been identified.

## Tasks:

| **Form the Modeling team** | **Project Manager** | **Required** |
|---|---|---|

The modeling team consists of the Domain Experts and the Chief Programmers along with other domain and development specialists, if any. Other developers are rotated throughout the domain modeling sessions in order to familiarize the developers to the project, domains and the process.

| **Conduct a Domain Walkthrough** | **Modeling Team** | **Required** |
|---|---|---|

A Domain Expert gives an overview of his domain. The walkthrough contains also information of domain that is not necessarily modeled or implemented. This phase familiarizes the Modeling Team to domain and the basic issues in it.

| **Study Documents** | **Modeling Team** | **Optional** |
|---|---|---|

The Modeling Team studies documents related to domain and requirements documents.

| **Develop Small Group Models** | **Modeling Team in groups** | **Required** |
|---|---|---|

In groups of three the Modeling Team members create a model of the domain area. The Chief Architect may give a “straw man” model to give a starting point for teams. Teams may also produce sequence diagrams to test and prove the proper function of the model.

| **Develop a Team Model** | **Modeling Team** | **Required** |
|---|---|---|

A member from each group presents the model developed by the group. The Chief Architect may also propose further model alternatives. The Modeling Team selects one of the proposed models or creates in collaboration a new model by combining ideas from the proposed models.

| | | |
|---|---|---|
| **Write Model Notes** | **Chief Architect, Chief Programmers** | **Required** |

Notes about modeling session are written. These should include notes on detailed or complex model shapes and on significant alternative models proposed. These are for future reference.

## Verification:

| | | |
|---|---|---|
| **Internal and External Assessment** | **Modeling Team, Stakeholders** | **Required** |

Domain Experts provide internal or self-assessment by actively participating in the subprocess. External assessment is done if needed by domain experts and customers through reviews and by requesting more detailed information on the domain.

## Exit criteria:

In order to exit the subprocess, an overall object model must be produced and accepted by the Chief Architect. This should show the architecture, classes and their connections, important attributes, some sequence diagrams explaining the harder parts and notes describing why this solution was selected and also showing the main alternatives.

### 4.2.2 Subprocess 2: Build a Features List

## Entry:

An initial project-wide phase that results in feature list needed to achieve the requirements. A team consisting of chief programmers is formed. Based on the partitioning of the domain in the Subprocess 1, the team breaks the domain into a number of areas (or major feature set). After this each area is broken into a number of activities (or feature sets). Each step within an activity is identified as a feature. The outcome of this is a hierarchically categorized list of features.

**Before Entry:** An overall object model has been created.

## Tasks:

| | | |
|---|---|---|
| **Form the Features List Team** | **Project Manager, development manager** | **Required** |

The features list team consists of the Chief programmers from the modeling team.

| | | |
|---|---|---|
| **Build the Features List** | **Features List Team** | **Required** |

The features are identified based on the first subprocess. Also functional requirements, user guides and other existing references to identify the features.

The aim of this task is a functional decomposition, breaking the domain into areas and the areas into activities that are composed from features representing a step in activity.

Features are simple granular client-valued functions. The features use a naming template: <action> <result> <object> (e.g. “Enter the desired number by dial pad”).

The granularity of a feature means that it should not take longer than two weeks to complete, but it should not be so granular that it would be simple getter or setter function. Steps that seem to take more than two weeks should be broken into smaller steps that become features.

## Verification:

| | | |
|---|---|---|
| **Internal and External Assessment** | **Features List team, Stakeholders** | **Required** |

The Features List Team participates actively in the process to provide internal or self-assessment. External assessment is done if needed by domain experts and customers through reviews and by requesting more detailed information on domain and on issues affecting the features list.

## Exit criteria:

In order to exit the subprocess, a feature list must be produced and accepted by the project manager and development manager. This should show the areas, activities within them and a list of features to accomplish each activity.

### 4.2.3 Subprocess 3: Plan by Feature

## Entry:

An initial project-wide phase that results in the development plan. The project manager, development manager and the chief programmers plan the order of the features to be implemented. The order of tasks in this subprocess is not strict but the tasks are more done together refining and considering tasks parallel.

**Before Entry:** The feature list has been created.

## Tasks:

| **Form the Planning team** | **Project Manager** | **Required** |
|---|---|---|

The planning team consists of project manager, development manager and chief programmers.

| **Determine the development sequence** | **Planning team** | **Required** |
|---|---|---|

The planning team assigns a completion date (month and year only) for each activity. The identification of the date is based on:

- Dependencies between features and classes involved in them,
- Balancing the load across class owners,
- Feature complexity,
- High risk and complex features should be first ones to implement, and
- Also external milestones should be considered (betas, previews, feedback, whole releases).

| **Assign Feature Sets to Chief Programmers** | **Planning Team** | **Required** |
|---|---|---|

Chief programmers are assigned as owners of the activities (or feature sets). This assignment is based on:

- Development sequence,
- Dependencies between features and classes involved in them,
- Balancing the load across class owners (the chief programmers are also class owners), and
- Feature complexity.

| **Assign Classes to Developers** | **Planning Team** | **Required** |
|---|---|---|

The Planning Team assigns developers as class owners. This is done based on:

- Balancing load across developers,
- Class complexity,
- The usage of the classes, and
- The development sequence

## Verification:

| **Internal and External Assessment** | **Planning Team, Stakeholders** | **Required** |
|---|---|---|

The Planning Team provides internal or self-assessment by actively participating in the subprocess.

### Exit criteria:

In order to exit the subprocess, development plan must be produced and accepted by the Project Manager and Development Manager. This should show the feature sets with completion dates and the chief programmers assigned to each of these features. The development plan also shows the assigned class owners.

## 4.2.4 Subprocess 4: Design by Feature

### Entry:

A per-feature activity to produce the feature design package. The features are scheduled for development by assigning them to a Chief programmer. The Chief programmer selects features for development from the features assigned for him.

The Chief programmer then forms a feature team by identifying the owners of the classes likely to be involved in the development of the feature. This team produces detailed sequence diagrams for the selected features. The Chief programmer then refines the object model and the developers write class and method prologues.

**Before Entry:** The planning team has successfully completed Subprocess 3 (Plan by feature)

### Tasks:

| **Form a feature team** | **Chief programmer** | **Required** |
|---|---|---|

The Chief programmer identifies the classes that are likely to be involved in the design of selected features and identifies the owners of these classes. The Chief programmer also starts new design packages for the selected features.

| **Conduct a domain walkthrough** | **Domain expert** | **Optional** |
|---|---|---|

The Chief programmer may request a Domain expert to walk the feature team through details (e.g. algorithms, rules, formulas, data elements etc.) of the selected features. This task is optional and is based on the complexity of the features or interactions.

| **Study the referenced documents** | **Feature team** | **Optional** |
|---|---|---|

The feature team studies the documents referenced in the features list for the features to be designed and any other useful documents, including memos, screen designs, hardware interfaces and external system interface specifications. This task is optional and is based on the complexity of the features or interactions and the existence of such documents.

| **Develop the sequence diagrams** | **Feature team** | **Required** |
|---|---|---|

The feature team develops detailed sequence diagrams required for each feature being designed. The team writes up and records any alternative designs, design decisions, assumptions, requirements clarifications and notes in the design alternatives or notes section of the design package.

| **Refine the object model** | **Chief programmer** | **Required** |
|---|---|---|

The Chief programmer creates a feature team area for the features. This area is either a directory on a file server, a directory on Chief programmer's computer or a directory at project's version control system. This area is used for the team to share its progress and make the progress visible.

The Chief programmer refines the overall object model to add additional classes, operations or attributes, based on the sequence diagrams defined for the features. The associated implementation language source files are updated in the feature team area. The model diagrams are created in publishable format.

| **Write class and method prologue** | **Feature team** | **Required** |
|---|---|---|

Using the updated implementation language source files from previous task, each class owner writes the class and method prologues for each item defined by the feature and sequence diagrams. This includes parameter types, return types, exceptions and messages.

| Design inspection | Feature team | Required |
|---|---|---|

The feature team performs a design inspection. Other project members may participate. The Chief programmer makes the decision to inspect within the feature team or with other project team members.

## Verification:

| Design inspection | Feature team | Required |
|---|---|---|

A successful design inspection is the verification of the output of this sub process.

## Exit criteria:

In order to exit the subprocess, a successfully inspected design package should be created. This should include a memo that integrates and describes the design package so that it can be reviewed independently, the referenced requirements, the sequence diagrams, design alternatives, the refined object model, generated output from the prologues and the to-do task-list entries for tasks on affected classes for each team member.

### 4.2.5 Subprocess 5: Build by Feature

## Entry:

A per-feature activity to produce a complete client-valued function (feature). The class owners implement the items necessary for their classes to support the design for the selected features. The code developed is then unit-tested and the code is inspected. The order of the development, unit testing and inspections is determined by the Chief programmer. After these have been completed, the code is promoted to the build.

**Before Entry:** The feature team has successfully completed Subprocess 4 (Design by feature)

## Tasks:

| Implement classes and methods | Feature team | Required |
|---|---|---|

The class owners implement the items needed to satisfy the requirements on their classes for the selected features. This includes development of any unit-testing code needed.

| Conduct a code inspection | Feature team | Required |
|---|---|---|

The feature team conducts a code inspection, either before or after the unit test task. The Chief programmer decides whether to inspect within feature team or with other project team members.

| Unit test | Feature team | Required |
|---|---|---|

Each class owner tests their code to ensure that all requirements on their classes for the selected features are satisfied. The Chief programmer determines what feature team-level unit testing is required, so what acceptance testing is done.

| Promote to the build | Chief programmer, Feature team | Required |
|---|---|---|

Classes can be promoted to build after successful code inspection and unit testing. The Chief programmer is the integration point for the entire features and responsible for tracking the promotion of the classes involved.

## Verification:

| Code inspection and unit test | Chief programmer, Feature team | Required |
|---|---|---|

A successful code inspection and the successful completion of unit tests is the verification of the output of this sub process.

### Exit criteria:

In order to exit the sub process, the feature team must complete the development of one or more whole features (client-valued functions). To do this it must have been inspected, unit tested and the code promoted to build.

## 4.2.6 Subprocess 6: Feature retrospective

### Entry:

A per-feature activity to improve day to day processes. The chief programmer or selected class owner prepares and facilitates a short retrospective session (no more than one hour). Outcome should be improvement to personal working habits, insights on software design or some input to milestone retrospective.

**Before Entry:** The feature team has successfully completed Subprocess 5 (Design by feature). A feature is completed, inspected, unit tested and promoted to code. Whole feature team is available for retrospective session

### Tasks:

| **Prepare session** | **Selected facilitator** | **Required** |
|---|---|---|

The facilitator plans the whole session and selects suitable way to observe the feature implementation. Facilitator also decides the minimal goal for the session.

| **Retrospective session** | **Feature team, Chief programmer** | **Required** |
|---|---|---|

The feature team goes through a short retrospective session where they gather data on the last task and generate insights on that. Finally the actions are decided and documented using at most one A3 sized paper.

| | | |
|---|---|---|
| **Reporting** | **Chief programmer, session facilitator** | **Optional** |

Findings from the retrospective session are reported to other Chief programmers and to the Project manager. The session results are used as input to Milestone retrospective.

### Verification:

| | | |
|---|---|---|
| **Self-inspection** | **Chief programmer, Feature team** | **Required** |

A successful retrospective session needs that everyone shares their feelings and knowledge. Each feature team member is responsible for participating openly and honestly to retrospective.

### Exit criteria:

In order to exit the subprocess, the feature team must come up with at least one personal process improvement idea, input to the Milestone retrospective or improvement to overall process.

## 4.2.7 Subprocess 7: Milestone retrospective

### Entry:

An activity performed when a milestone is reached in order to improve processes and working environment. The Project manager or selected person prepares and facilitates a retrospective session (no more than one work day). Outcome should be improvement to process, insights on software design and general improvement ideas.

**Before Entry:** All the features needed to complete milestone have been developed and Feature retrospectives have been performed. Everyone who has participated implementation of the features belonging to this milestone is available for retrospective session

## Tasks:

| | | |
|---|---|---|
| **Prepare session** | **Selected facilitator** | **Required** |

The facilitator plans the whole session and selects suitable way to observe the feature implementation. Facilitator also decides the minimal goal for the session and what issues to emphasize during this session.

| | | |
|---|---|---|
| **Retrospective session** | **Each feature teams, everyone involved in development, Project manager** | **Required** |

The feature team goes through a standard retrospective session according to facilitator's plans. The project members gather data on the last milestone and generate insights on that. Finally the actions are decided and documented using at most one A3 sized paper. All inputs from Feature retrospectives are handled and actions from previous Milestone retrospective are reflected. Previous actions can be discarded, continued or adjusted.

| | | |
|---|---|---|
| **Reporting and implementation** | **Project manager, Chief programmers** | **Required** |

Findings from the retrospective session are documented and implementation is planned. The session results are always implemented and followed for at least few features forward.

## Verification:

| | | |
|---|---|---|
| **Self-inspection** | **Project Manager, Chief** | **Required** |

| | | |
|---|---|---|
| | **programmers, development team members** | |

A successful retrospective session needs that everyone shares their feelings and knowledge. Each feature team member is responsible for participating openly and honestly to retrospective.

Project manager and chief programmers verify that all the decided actions are implemented and possible problems are solved and communicated immediately.

### Exit criteria:

In order to exit the subprocess, the whole development team must come up with at least one process improvement action or a general improvement action.

## *4.3 Practices used with process*

Practices are common day-to-day activities [Beck05]. They should be clear and objective e.g. you can easily say that your company is practicing individual code ownership or collective code ownership. The software development processes are usually built around a set of best practices. The practices usually aren't new, but the combination of these may be. Each practice should support other practices and make working easier. The benefit resulting from a combination of suitable practices is greater than the sum of benefit from the individual practices [Palm02].

The proposed process has also practices defined with it. These practices are built on the values and principles from the agile manifesto. Each of the practices should support software development efforts and help the company deliver client-valued software. These practices can be extended, can be replaced with other practices that suit the needs better and can be removed.

The base practices are the eight practices from the feature driven development [Palm02]:

- Configuration management
- Developing by feature
- Domain object modeling
- Feature teams
- Individual class ownership
- Inspections
- Regular builds
- Reporting/Visibility of results
- Unit testing

Few agile modeling practices are also used [Amb02]:

- Active stakeholder participation
- Apply the right artifacts
- Consider testability
- Create several models in parallel
- Model with others
- Use the simplest tools

Also few practices have been taken from Scrum and Extreme programming:

- Energized work [Beck05]
- Retrospectives [Schw01]

The following paragraphs describe briefly each of the practice listed above and also few benefits resulting from following the practices. Some practice dependencies are also mentioned.

### 4.3.1 Feature-Driven Development practices

The following FDD practices form the core practices used in the process.

**Configuration management (FDD practice)**

The configuration management in an FDD project should be able to identify what features have been completed to date and to maintain change history of all classes. These needs vary greatly depending on project's demands and the complexity of the product to be developed. [Palm02]

A very important point is that the configuration management system should not only hold source code files, but also requirements documents, analysis and design artifacts, test cases, test harnesses and scripts and test results should be in version control system. Even the development tools could be version controlled. [Palm02]

**Developing by feature (FDD practice)**

In FDD functional requirements are expressed as features. One feature is one client-valued function defined with language that the business side can understand. Features are a functional decomposition of the requirements in a very similar way to stories, use cases and use case diagrams. These features are used to track and steer development. The customers are able to prioritize features according to their business value. [Palm02]

The main points about the features are that features have to be small and client-valued. The size limit for a feature is that it should be implemented in two weeks. Usually the project team should aim for features with smaller granularity, from few hours to few days. Small size enables better and more precise tracking of progress on frequent basis. [Palm02]

Client-valued means that each feature should map into some kind of process, be it typical business process or a process to set up a lawnmower robot. These features are specified in a certain template:

<action> <result> <object>

One example from this could be "Perform a cleanup on the lawnmower robot". This tells also a bit about the implementation of the feature, at least a function called performCleanup() should be in Robot class. [Palm02]

**Domain object modeling (FDD practice)**

Domain object modeling is modeling the problem domain with class diagram. This shows the significant objects, the services they offer and the relationships between the objects. Also e.g. some high level sequence diagrams may help developers to understand the problem domain, so there should be considerations on usage of multiple models in the domain object modeling. [Palm02]

The first purpose of domain object modeling is to bring the assumptions made by individuals on the table. After learning the requirements, the developers tend to make some automatic assumptions and bringing them open is important so that every assumption is handled and no false assumptions remain. The aim is for common understanding inside the development team and minimal amount of misunderstandings. [Palm02]

Palmer uses a metaphor of road map to describe the domain object model [Palm02] compared to the driving metaphor by Kent Beck [Beck05]. As Beck says, building software is like driving a car. You observe your environment and according to it you steer, accelerate and break. The domain object model is a road map for the driver. He knows which way he should go and he gets there faster and without so many detours. [Palm02], [Beck05]

The domain object model is an overall framework, where you add more classes, functions, and attributes, feature by feature. Each developer should have an overall view on the system and the domain object model helps to maintain the conceptual integrity. The amount of needed refactoring should be minimized by using the domain object model to guide the implementation. [Palm02]

**Feature teams (FDD practice)**

The feature teams are dynamically created teams that produce features. A feature team is started by the Chief programmer when he selects a new feature to be implemented next. The Chief programmer decides which classes are needed to implement the feature and contacts the Class owners. The Chief programmer and the Class owners involved in this feature are the feature team, like presented in Figure 27. This dynamic team formation makes Feature Driven Development highly scalable. [Palm02]

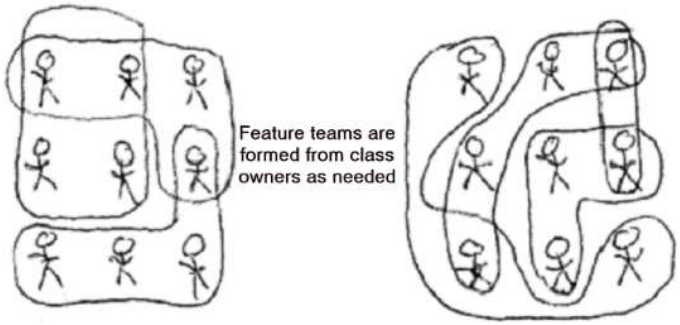


**Figure 27 Feature team formation.**

In order to feature teams to work effectively few other practices are needed. First of all individual class ownership is needed to identify the Class owners that are needed in the feature team. Also domain object modeling is needed as most of the classes should be known before starting the development of a feature. [Palm02]

The feature teams are usually quite small, from three to six developers. The feature team members own all the code that is needed to be changed in order to implement the feature under work. So there are no team dependencies for implementation of one feature. The Chief programmers are also class owners, so they keep working with the source code. Also a Class owner may belong to multiple feature teams at the same time. [Palm02]

**Individual class ownership (FDD practice)**

Feature driven development has a bit different view on code ownership. Each class is owned by a Class owner. This makes one developer responsible for the conceptual integrity of the class. A Class owner is also an expert on this one class and can explain the purpose and function of some piece of code. A Class owner also implements new

functionality to the class faster than someone who has not worked with the class before. Also individual class ownership gives each developer something to be proud of and see the products of their own effort. [Palm02]

**Inspections (FDD practice)**

Code and design inspections have proved to be effective way to find defects from software. The bad reputation of inspections is mainly because inspections are often done quite badly. Idea of inspections is to let multiple people look through work artifacts to ensure code quality, to transfer knowledge and to see that the artifact is according to standards. One important point in inspections is that it should not be about individual performance but team performance. There should be a lot of effort on making the inspections more comfortable for individuals and creating an atmosphere where inspections can be seen as a good thing. [Palm02]

**Regular builds (FDD practice)**

One important aspect in most agile methodologies is regular or even continuous builds. The system should be build at regular intervals, be it weekly, daily or after every source code commit to configuration management system. This is because one traditional pain in waterfall kind of software development has been integration or build phase. By pushing this phase to the end of the project the project team can be certain that there will be major problems and costly fixing. The more often build process is done the more early the integration problems will arise. [Palm02]

Regular builds also ensure that there is always a working version of the software for demonstration purposes. Build process can be used also to create documentation, run unit test suites, run code metric collection, test static code quality and run acceptance tests. In Extreme programming continuous integration is one core practices. This means that all code is build each time anyone commits code to configuration management system, meaning that the build process is run every hour. The build process is also defined so that it should produce the final product. So if the software is a web site then

the web site should be deployed, and in case of a embedded software, it should be flashed to hardware. [Palm02], [Beck05]

**Reporting/Visibility of results (FDD practice)**

Normal problem with waterfall processes is its lack of visibility or even false understanding of progress. Feature driven development is very strong in reporting the current status of the project. Each feature can be tracked and different states of features are precisely defined. Also it is easy to track the number of features completed. [Palm02]

**Unit testing (FDD practice)**

Unit testing is a testing practice that tests the software in small units separated from other units. In procedural programming languages this unit might be e.g. a function and in object oriented languages the small unit is a class. The unit tests are usually written in the same programming language as the unit under test and are usually written by developers. [Hunt03]

Testing units separated from other units means that a test is not a unit test if

- it uses the database,
- it communicates across the network,
- it uses the file system,
- it can not be run in parallel with other unit tests, and
- you have to do manual work to run it, e.g. edit config files, set the system to certain state etc.
  [Kos08]

Unit testing gives multiple benefits to developers. Comprehensive unit test suites enable developer to refactor code and implement new changes without a fear of breaking the software from other parts. Running the test suites will show if any problems result from the changes. Also the unit tests offer an example on how to use class or function under

test. So in a sense it works as an executable documentation as well as a safety net for changes. [Mart03]

### 4.3.2 Agile Modeling practices

Some Agile Modeling practices were selected for the process in order to bring the Extreme Programming values and principles as a part of the development process. These were also seen as suitable practices for the target organization as it sees modeling as an important factor in software development projects and, as described in Section 3.3.6, there are quite many issues related to modeling and design.

**Active stakeholder participation (AM core practice)**

The Active stakeholder participation practice is similar to On-site Customer [Beck05] practices found from Extreme programming. A project stakeholder is someone who is a user, a manager of users, a support staff member, a tester or anyone who is affected by development of the software. Project success is often tightly linked to the participation of important stakeholders. Management needs to support the project, operation and support staff needs to work with the project team to make your environment ready, hardware designers need to work with developers to make the system function as good as possible, users and customers need to give feedback on development efforts so far and give business knowledge and clarifications to the project team. [Amb02]

Active stakeholder participation was seen as needed practice in order to make sure the stakeholders have a possibility to give feedback to the team. In a sense the practices was already part of the FDD subprocesses.

Active stakeholder participation is done by first of all using stakeholders as Domain experts in domain walkthroughs. Also project management support is needed, so the project team has to make sure the management understands the business opportunities and value of the product. Also the management has to understand the benefits of the

used techniques and technologies and the decision leading to usage of these. In order to gain this knowledge, the project team has to get management to participate in the project. [Amb02], [Amb05]

**Apply the right artifacts (AM core practice)**

Apply the right artifacts practice is all about using the right tool for the job. In case of modeling, this means that correct artifact, be it a UML state chart, source code, a data flow diagram etc, is used when it is most appropriate. Each of these artifacts has their own strengths and weaknesses and people are more competent with different artifacts. The correct artifact to fit different situations should be selected. In order to master the differences between different artifacts, people need to be trained and the more experienced developers should be listened when selecting suitable artifact for the job. [Amb02], [Amb05]

This practice was selected as it directly helps with one specific issue described in Section 3.3.5. The apply right artifacts practice reminds the developers not to model with just one artifact, but to model the problem domain from suitable and needed viewpoints.

**Consider testability (AM core practice)**

‘Consider testability’ is very important practice when developing software for embedded systems. The final hardware may not be available until the end of the project and if the software can not be tested before that, the project will be in trouble. This was seen as a large problem in Sections 3.3.1 and 3.3.2. Also the lack of testability was part of the issues described in Section 3.3.3.

The practice states that all the time during modeling, you should ask yourself "How are we going to test this?" No non-testable software should be build. One mantra of agile developers is "test early and test often", meaning that the developers make sure that they are doing the work right and to get immediate feedback to notice mistakes. [Amb02]

One use of this practice is during the "Develop overall model" subprocess. The model should have defined hardware interfaces that can be used when testing the software using mock objects [Hunt03].

**Create several models in parallel (AM core practice)**

As each different artifact has its own strengths and weaknesses, you may need multiple models to succeed in modeling. In FDD this practice is followed at least in some extent, as features are modeled with sequence diagrams and with the overall object model. [Palm02]. However these two diagrams may not be enough, so usually e.g. UI prototypes, use cases, and especially in embedded systems state charts are needed. When modeling, developers must remember to use the right artifacts for the job and to create multiple models when needed. [Amb02]

This practice helps with the same issue than the Apply the right artifacts practice described above and will address the issues described in Section 3.3.5.

**Model with others (AM core practice)**

Software development and modeling are creative tasks and very error prone. In fact when modeling, you can not be certain that your design will work. Because of these the modeling should be performed in a small group. This makes it easier to toss out ideas, divide knowledge and efficiently better the models under work. As a result from a collaborative modeling session you should not only have better design, but also better common knowledge, vocabulary and vision of the system. This practice comes right from the agile values enhancing communication. [Amb02]

Model with others practice promotes communication which also is one of the main values in Agile manifesto as described in Section 2.2. The practice was selected to promote team work and to accelerate learning in project work.

**Use the simplest tools (AM core practice)**

Modeling is a task that is more or less about evolving the design in your head and communicating it to others for feedback and elaboration. Hence it usually should be done with very simple, easy-to-use and effective tools. Use the simplest tool practice requires developers to use the simplest tools suitable for the task at hand. Usually a whiteboard or a large sheet of paper is enough, but also CASE (Computer-aided software engineering) tools are sometimes the simplest suitable tools. [Amb02]

This practice also spreads the value of simplicity which is one of the core values in Extreme Programming [Beck05]. This was one purpose for selecting this specific practice to be part of the process. It also helps out the issues of complexity described in Section 3.3.6 and the issues with suitable tools described in Section 3.3.3.

### 4.3.3 Other agile practices

The following practices were selected for the process from Extreme Programming and Scrum methodologies. Similar and suitable practices for the issues these practices address were no found from Agile Modeling or Feature-Driven Development.

**Energized work (XP primary practice)**

Energized work is one of the main practices in Extreme programming. It states that developers should work as long as they are productive and with sustainable pace. There is no point in overburdening developers with long hour and constant hurry. Running a marathon is usually considered as a metaphor for energized work practice. Marathon should be run on sustainable pace in order to finish. [Beck05]

The energized work practice directly addresses the issue of not having time for break described in Section 3.3.4.

**Retrospectives (SCRUM practice)**

The retrospectives are facilitated meetings where team looks back and tries to learn and improve for the future work. There are few main reasons for holding retrospectives. First of all of course the learning, a team can learn from problems and errors and also a team can try to repeat successes. Secondly retrospectives improve communication by bringing the team together and going through issues from personal view points of team members. Third reason for retrospectives is making the team a part of process improvement and at the same time making them more committed to the process used. It is more natural for people to commit on something they have created or altered by themselves. [Dav04]

The retrospectives aim for continuous improvement. This concept has been talked a lot because of gained popularity of Lean software development [Poppen03], [Poppen07]. The retrospectives are one way to achieve this improvement. The decisions and ideas made at retrospectives should be used so that the retrospectives do not feel like unnecessary waste of time.

The reason for selecting retrospectives as integral parts of the process was that we had concerns about how the FDD practice itself handles issues and promotes continuous improvement. The retrospective was an ideal choice for this and the outcome from retrospectives helps us address almost any issue that is slowing the development team down.

## *4.4 How the process meets the development issues?*

The following sections map the issues of traditional embedded software development to suitable practices that solve or relieve the corresponding issue. There is also a short description of the solution related to the issues. The mapping is done in the same order as in Section 3.3.

### 4.4.1 Response time and timing

| Issue | Practice |
|---|---|
| No HW available | Unit testing |
| No exec time measurements done | Early and frequent testing |
| No HW analysis done | Frequent releases of working software |
| No memory analysis done | Early and frequent testing |

**Figure 28 Response time and timing related issues with helping practices.**

The first issue of not having hardware available during software development can be eased with the practice of unit testing. It can be done against simulated or mocked hardware to ensure that the program logic is valid. This way the testing of hardware-software interface can be done focusing mainly on the interactions between these two parts and it is possible to test other parts of the system independently.

The issues of lacking execution time measurements and lacking memory analysis need early and frequent testing. This is done in the process by developing the software incrementally and testing each increment. By testing as soon as possible and as frequently as possible, the performance and execution time issues can be brought visible.

The missing hardware analysis can cause performance issues and compatibility issues. This can be helped with frequent releases of working software which is integrated with hardware and tested as a whole will reveal performance issues resulting from poor HW analysis. Again the incremental way of implementing the system drives the frequent releases.

### 4.4.2 Platform architecture

| Issue | Practice |
|---|---|
| Distributed and multiprocessor platforms | Unit testing, Early and Frequent testing |
| Load balancing | Early and frequent testing |

| | |
|---|---|
| Interprocess communication | Unit testing, Early and Frequent testing |
| Generalizations | Incremental development |
| Variable and unique platforms | Unit testing |
| Resource allocation and management | Early and frequent testing, unit testing |

**Figure 29 Platform architecture related issues with helping practices.**

Many of the issues with platform architecture can be solved partly with unit testing. It can be done against simulated hardware and distributed systems to ensure that the program logic is valid. This way the testing of hardware-software and network communication can be done focusing mainly on the interactions between these two parts. Platform issues, as well as load balancing and performance issues, can also be found through early and frequent testing done after each implemented increment.

Incremental development leads to evolving design which provides a possibility to do suitable generalizations when needed. Also proper unit testing enables us to isolate the non-changing parts of the system to own modules and makes it easier to hide the hardware platform behind a facade or interface.

### 4.4.3 Embedded software development tools

| **Issue** | **Practice** |
|---|---|
| Lacking features | Retrospectives, develop by feature |
| Selection process | Retrospectives, Incremental development |
| No automated testing | Retrospectives, Unit testing |

**Figure 30 Embedded software development tools related issues with helping practices.**

The retrospectives help the team to self assess and continuously improve. This also enables us to select our tools incrementally and evaluate tools while using them in the actual development work. Retrospectives also uncover our needs for better tools and e.g. automated testing.

The Develop by feature practice ensures that each feature is finished before moving to lower priority features. So these all makes it possible to select the tools while working

forward with the software. We can change development tools for future features, and keep the existing tools working with the previous features.
So the retrospectives and incremental development defer the selection of tools to latest responsible moment. Also tools can be changed after testing them in the project work.

### 4.4.4 Typical constraints

| Issue | Practice |
|---|---|
| Failure recovery | Plan by feature, early and frequent testing |
| Battery and processing power | Plan by feature, early and frequent testing |
| Interaction with the environment | Plan by feature, early and frequent testing |
| Updates, correctiveness | Plan by feature, early and frequent testing |

**Figure 31 Typical constraints related issues with helping practices.**

Plan by feature makes it sure that the highest priority features and critical issues are resolved first. This combined with early and frequent testing gives us feedback on how these solutions work and how to improve them. These two practices help us to work with the typical constraints and to bring the constraints visible as early as possible.

### 4.4.5 Development process

| Issue | Practice |
|---|---|
| No code reviews | Code and Design inspections |
| Documentation written too late | Included in design packages |
| No time for break | Energized work |
| Task switching | N/A |

**Figure 32 Development process related issues with helping practices.**

Code and Design inspections that are included in the development process ensure that code is always reviewed. Also the documentation is one deliverable in design packages, so it needs to be done in order to finish a feature.

There is no sense in overburdening the project team as the projects are usually long. The energized work practice instructs the teams to in work sustainable pace. However, FDD

gives multiple responsibilities to each developers and this might cause excessive amount of task switching occasionally. This can be seen as a downside of the process.

### 4.4.6 Programming practices

| Issue | Practice |
|---|---|
| Large if-then-else and case statements | Code and Design inspections |
| No naming and style conventions | Code and Design inspections, Retrospectives |
| Delays implemented as empty loops | Code and Design inspections |
| One big loop | Code and Design inspections |
| Global variables | Code and Design inspections |
| Workarounds | Code and Design inspections |
| Configuration information in #define statements | Code and Design inspections |
| Including #global.h | Code and Design inspections |
| Wrong use of interrupts | Code and Design inspections |

**Figure 33 Programming practices related issues with helping practices.**

Continuous peer review helps to catch technical problems in design and code level. These inspections also help to spread the knowledge about correct practices. The correct style and conventions can be agreed in the kick-off and changed in any retrospectives. Collaborative working with the code improves the code quality over time.

### 4.4.7 Design practices

| Issue | Practice |
|---|---|
| Dependencies | Design inspections, Unit testing |
| Only one design diagram | Concept Diagram, Sequence diagrams, apply right artifacts, create several models in parallel |
| One solution applied every time | Code and Design inspections |
| Overdesign | Code and Design inspections, Scoping, |

| | |
|---|---|
| | Unit testing |
| Reusing non-reusable modules | Design inspections |

**Figure 34 Design practices related issues with helping practices.**

Dependencies are a common issue in large embedded software. Dependencies makes unit testing painful, so having lots of dependencies is revealed when writing unit tests. Also design inspections reveal possible dependencies and are also a step that makes it possible to collaboratively improve the design.

Different diagram types are included into the process in order to produce a big picture about the whole problem domain as well as a picture of the functionality of certain feature. Also the Agile Modeling practices apply right artifacts and create several models in parallel reminds the developers not to use only one design diagram for everything.

Continuous attention to code and design inspections keeps the code simple and understandable as well as the design is cleaner. It also contributes into a common understanding of the project and the architecture. Unit testing is hard if the design is overly complicated so that drives also the simple design. Finally, the scoping helps to focus on one problem at a time instead of designing for all possible combinations at once.

### 4.4.8 Project size and complexity

| **Issue** | **Practice** |
|---|---|
| Too long projects | Prioritized requirements and release working software early and often |
| Natural complexity | Incremental development |

**Figure 35 Project size and complexity related issues with helping practices.**

The issue with long projects can be relieved by prioritizing the requirements and releasing working software early and often. This might shorten the project by scoping out the unnecessary features. According to Standish group studies, 45 % of the

requirements are rarely or never used [John02]. Also it adds new milestones for the project, and so divides it into smaller objectives.

The natural complexity can be eased by solving the problem in incremental steps, starting from the simplest case and building on top of that.

# 5 Retrospective of the process development

The target of the work was to develop an agile process for training and consulting company. The company had a major customer in Finland that had a need for agile methods, but was not sure what method would be suitable for their environment and how to adopt that. The customer was in developing consumer electronics for the global market. So the target for the developed process was to suit the needs of that specific target organization and to generate profit for the company. The company had also few good experiences from self developed processes from the previous decade.

The project started by a background study of the most popular agile methods and case studies of those. The analysis consisted mostly from estimating different aspects of the process and reflecting those to the target organization. All the methods were characterized into their stereotypes: XP was very radical, FDD feature oriented, Scrum just a framework, DSDM was for business and financial software etc.

There were many factors making the Feature-Driven Development a good choice for the base of the process. The target organization was quite traditional and process oriented. It could be described as a bit slow in changes. So the Extreme Programming, which was at that time the most popular method, was seen a bit too radical. Also the XP was seen as ad-hoc method for small team to do software. The FDD, on the other hand, was very process oriented and was quite popular. It was even ahead of Scrum in some surveys and was the runner up for XP.

Another interesting part of FDD is that it promises scalability. In the target organization, there was very large number of large projects, so this was a promise that was quite important in selection process. The terminology, especially Features, of FDD was also familiar to the target organization.

Another part of the process was the Agile Modeling practices. We selected Agile modeling as a sidekick for the FDD in order to bring in the XP and agile values to the target organization. The AM is also an easy and simple process to implement and to spread inside an organization. Also it was a natural selection for this as modeling was seen as a very important activity in target organization and the projects were quite model driven.

After the overall view of the agile methodologies was done and suitable candidates were found, we started to investigate the special characteristics of embedded software development and the development projects. From the technical side, we quickly identified that the largest difference to traditional software development was the close interaction of hardware and software. Also common was to have customized hardware that would change from a project to another.

Looking at the embedded software development projects, we identified that the projects were often very large and had a heavy and very structured organization. To handle this large amount of people and large organization, the projects were very process oriented and risk driven.

After the larger overview of embedded software development, we decided to identify the main issues of the embedded projects from literature and to build the new process to somehow help out with these issues and improve the current situation.

A while after the process development work had started we had our first doubts about the process under work. FDD was very process oriented and in a sense it lost some of the main points of agile underneath the process. Emphasis on communication, continuous improvement and team work was not brought up in the FDD process. Also on development side, the feedback was very hard to get.

Some FDD practices were completely opposite from other popular agile methods. FDD tracks process of features through describing a percentage of work already done, instead of measuring only done software as stated already in the agile manifesto.

The modeling was a big part of the FDD process and the whole system is roughly illustrated before anything is done. This can be seen like a big design up front (BDUF) that has been treated as a harmful practice in Extreme Programming.

Another practice that contradicts with XP is the individual code ownership in FDD. FDD states that each class should be owned by one developer, when the XP states that all code should be common and everyone should be able to make changes anywhere.

Finally, by combining the individual code ownership and feature teams, FDD creates a big amount of task switching for developers. Task switching has been seen as a bad habit in agile development and many methodologies emphasize on getting one thing done at a time.

Some changes were done to the process at this point. The most notable was the addition of retrospectives to the process. This was done to ensure the feedback for the developers and to enable continuous improvement.

The process development ended at the beginning of the 2007. It was partly due to other, more valuable work and also due to the lack of interest from the target organization. There were quite many reasons behind this, but we try to describe our observations here.

When the process development work started, the FDD and Scrum were almost equally popular methodologies. However, during the development Scrum gained popularity and was overcoming XP. At the same time the community was losing interest to FDD. This change in popularity happened in a short time and it seems to be quite common that processes and methodologies have their own lifecycle. Some will gain larger popularity and become standards in the industry as others will have a small popularity that fades away in time.

Another movement we noticed during the process development was the increase of the amount of published agile research. The industry had more experiences about agile and the researchers were all the time producing more information about the agile development. Most of these researches were about Scrum and XP. It was quite hard to find any FDD related publications.

As the Scrum popularity was rising, we also noticed that there was no need for specific own process, but rather a framework that could be extended was seen more valuable. We could not see anymore the possibilities of creating profit by selling the developed process and trainings related to that. Scrum was seen as an easier way to create profit for the company, so we decided to prioritize our Scrum related products over the own developed process.

The one aspect that was not considered in this development work was that no process fits for all projects. Each project, even inside one company, is unique and the adoption work and modifications have to be made for each project. So it was not likely that the developed process would be suitable to all needs, and the modifications of the process would have been more complex than for example when adopting Scrum.

We did find out few important lessons while working through the long process of creating the modified version of FDD.

- No process is useful on paper
- No process can be made without engaging the people
- Faster delivery increases chances of success

We had made quite solid work with the process and were quite happy with the outcome. After the development work ended and there was no one using the process, we noticed that there is no value in having a purely theoretical software development process. In order to generate value with a process, it must be used to develop software. This would have also helped to identify the shortcomings of the process.

For this process, we believe that through better personal engagement and commitment the process might have had an empirical test. So in order to bring the process to the daily work a certain amount of leadership is needed.

We also noticed that the process development was very hard, if started just from the issues the developers have. We had no idea of the biggest problems the developers faced in the day to day work. We had a vague idea of the most common issues in the industry and we started solving those.

After the process was developed, it was obvious that the lean way of having the people who work with that process should have been the people who contribute to the process. This would have given us new and diverse perspectives as well as added commitment and buy in from the people who start working with the new process.

We decided in the beginning to make a process that would be ready and easy to adopt. There was no talk about milestones or early releases. However, we believe that by delivering the first drafts of the process as early as possible and after that frequently delivering the changes, we might have improved our chances of success greatly.

# 6 Summary

The thesis presents only theoretical basis for embedded software development process. This process should be evaluated in practice in order to give value to the embedded software development community. This could be one possible aspect for further studies.

The more valuable part of this thesis is the new practices customized into Feature Driven Development. Especially retrospective is commonly thought as a highly important practice and should be adopted in order to gain benefit from continuous improvement.

The thesis project was quite a long one, due to business environment changes, and due to personal day to day work demands. During this time the agile development methods have become quite common in software development projects. Especially Scrum has become highly popular development framework. All in all the agile software development has crossed the chasm. There still is need to research the different flavors of agile in embedded system projects.

The process presented in this thesis does not describe anything about adoption and change. Agile adoption has been seen as quite painful process and as such requires good guidance, lots of knowledge and effective communication. In order to adopt a process a much more is needed than a process documentation and detailed day to day practice descriptions. This has been left out from this thesis and could be another study that should be conducted.

The key findings from this thesis are related to the process of developing a new software development process. These can be summarized into following three observations:

- No process is useful on paper
- No process can be made without engaging the people
- Faster delivery increases chances of success

# References


[Abra02] Abrahamsson, Pekka et al. 2002. Agile software development methods: Review and analysis. <http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf>

[Amb02] Ambler, Scott 2002. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. John Wiley & Sons, Inc.

[Amb04] Ambler, Scott; Jeffries, Ron 2005. The Object Primer, Third Edition: Agile Model-Driven Development with UML 2.0. Cambridge University Press

[Amb05] Ambler, Scott 2005. The Agile Unified Process (AUP) Home Page. <http://www.ambysoft.com/unifiedprocess/agileUP.html> 12.6.2006

[Amb06] Ambler, Scott 2006. Agile Survey Results Summary <http://www.ambysoft.com/downloads/surveys/AgileAdoptionRates.ppt> March 2006

[Amb07a] Ambler, Scott 2005-2007. An Introduction to Agile Modeling. <http://www.agilemodeling.com/essays/introductionToAM.htm> 30.6.2008

[Amb07b] Ambler, Scott 2007. Survey Says... Agile Has Crossed the Chasm - Examining the effectiveness of agile practices. Dr. Dobbs Journal August 2007, 59–61

[Amb07c] Ambler, Scott 2007. Agile Survey Results Summary <http://www.ambysoft.com/downloads/surveys/AgileAdoption2007.ppt> March 2007

[Amb08] Ambler, Scott 2008. Agile Survey Results Summary <http://www.ambysoft.com/downloads/surveys/AgileAdoption2008.ppt> February 2008

[Ang06] Angel, Justin-Josef 2006. Feature Driven Development: For the agile agent of change.

<http://www.JustinAngel.Net/files/FeatureDrivenDevelopment.ppt> 29.6.2006

[Awad96] Awad, Maher; Kuusela, Juha; Ziegler, Jurgen 1996. Object-Oriented Technology for Real-Time Systems. Prentice Hall

[Barr07] Barr, Michael; Massa, Anthony 1999,2007. Programming Embedded Systems with C and GNU Development Tools. O'Reilly

[Bay94] Bayer, Sam; Highsmith, James A. 1994. RADical software development. American Programmer 7(6): 35-42.

[Beck99] Beck, Kent 1999. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional

[Beck05] Beck, Kent; Anders, Cynthia 2005. Extreme Programming Explained, Second Edition: Embrace Change. Addison-Wesley

[Ber07] Bergman, Gustav; Coplien, Jim. Is Something Rotten in the Practices of XP? Lean Magazine December 2007. Softhouse Nordic

[Boehm00] Boehm, Barry 2000. Project Termination doesn't Equal Project Failure. Computer, Volume 33, Issue 9, Pages 94-96. September 2000.

[Boehm03] Boehm, Barry; Turner, Richard 2003. Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Professional

[Bro75] Brooks Jr., Frederick P 1975, 1995. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley.

[CMP01] CMP Media LLC, Embedded Systems Programming, Wilson Research Group 2001. 2001 Embedded Market Study. Conducted August-September 2001.

[CMP02] CMP Media LLC, Wilson Research Group 2002 Embedded Market Study. Conducted October-November 2002.

[CMP04] CMP Media LLC, Wilson Research Group 2004. 2004 Embedded Market Study. Conducted November 2003 – January 2004.

[CMP05] CMP Media LLC, Wilson Research Group 2005. 2005 Embedded Market Study. Conducted February-March 2005.

[CMP06] CMP Media LLC, EE Times, Embedded Systems Design 2006. 2006 State of Embedded Market Survey. Conducted March 2006

[Cock01] Cockburn, Alistair 2001. Crystal light methods. <http://alistair.cockburn.us/index.php/Crystal_light_methods>

[Cock02] Cockburn, Alistair 2002. Agile Software Development. Pearson Education, Inc.

[Coad99] Coad, Peter; Lefebvre, Eric; De Luca, Jeff 1999. Java Modeling in Color with UML. Prentice Hall

[Dav04] Davies, Rachel; Brown, Peter. Using Retrospectives to Channel Feedback <http://agilexp.com/presentations/XPDay4Retrospectives.ppt> 2004

[DeM87] De Marco, Tom; Listener, Timothy 1987,1999. Peopleware: Productive Projects and Teams. Dorset House Publishing.

[Dru07] Druckman, Angel 2007. What Scrum Can and Cannot Fix? Scrum Alliance. <http://www.scrumalliance.org/articles/68-what-scrum-can-and-cannot-fix>

[DSDM03] DSDM Consortium 2003. DSDM Public Version 4.2 Manual <http://www.dsdm.org/version4/2/public/>

[DSMD08] DSDM Consortium 2009. DSDM Atern. <http://www.dsdm.org/atern/>

[Eck04] Eckstein, Jutta 2004. Agile Software Development in the Large: Diving Into the Deep. Dorset House Publishing Company

[Els07] Elssamadisy, Amr 2007. Patterns of Agile Practice Adoption - The Technical Cluster. C4Media Inc

[Emb06] Zurawski, Richard 2006. Embedded Systems Handbook. Taylor & Francis Group, LLC.

[Emb08] 2008 Embedded Systems Market. Tech Insight/Embedded Systems Design

[Fowl05] Fowler, Martin 2005. The New Methodology. <http://www.martinfowler.com/articles/newMethodology.html>

[Fre09] French Scrum User Group 2009. A National Survey on Agile Methods in France. June, 2009. <http://www.frenchsug.org/download/attachments/591296/National_survey_FrenchSUG_ENGL_en.pdf?version=2> 27.5.2010

[Gan01] Ganssle, Jack G 2001. A Guide to Code Inspections. The Ganssle Group. <http://www.ganssle.com/inspections.pdf>

[Haa07] Haapio, Petri 2007. The Agile Change in NSN. A seminar presentation in Agile Finland Fall 2007 seminar 3.10.2007

[High00] Highsmith, James A. 2000. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House Publishing

[Hunt99] Hunt, Andrew; Thomas, David 1999. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley

[Hunt03] Hunt, Andrew; Thomas, David 2003. Pragmatic Unit Testing: In Java with JUnit. The Pragmatic Programmers, LLC

[Info06] Hartman, Deborah 2006. Interview: Jim Johnson of the Standish Group. <http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS>.

[John02] Johnson, Jim 2002. XP 2002 conference keynote speech.

[Jones97] Jones, Capers 1997. Software Quality - Analysis and Guidelines for Success. International Thomson Computer Press.

[Jones04] Jones, Capers 2004. Software Project Management Practices: Failure Versus Success. <http://www.stsc.hill.af.mil/crosstalk/2004/10/0410Jones.html>

[Kni07] Kniberg, Henrik 2007. Scrum and XP from the Trenches - how we do Scrum. Lulu.com

[Koop96] Koopman, Philip 1996. Embedded System Design Issues (the Rest of the Story). Proceedings of the International Conference on Computer Design 1996.

[Kos08] Koskela, Lasse 2008. Test Driven. Manning

[Kra03] Krasner, Jerry 2003. Embedded Software Development Issues And Challenges - Failure Is NOT Optional - It Comes Bundled With The Software. Embedded Market Forecasters. <http://embeddedforecast.com/emf_esdi&c.pdf>

[Kru00] Kruchten, Philippe 2000. The Rational Unified Process: An Introduction. Addison-Wesley Professional

[Lef07] Leffingwell, Dean 2007. Scaling Software Agility: Best Practices for Large Enterprises . Addison-Wesley Professional

[Lik04] Liker, Jeffrey K. 2004. The Toyota Way - 14 Management Principles from the World's Greatest Manufacturer. McGraw-Hill

[Man01] Beck, Kent; Beedle, Mike, van Bennekum, Arie et. al. 2001. Manifesto for Agile Software Development. <http://agilemanifesto.org/>

[Mart98] Martinez, Michael E. 1998. What Is Problem Solving? Phi Delta Kappan, Vol. 79, 1998

[Mart03] Martin, Robert C. 2003. Agile Software Development: Principles, Patterns and Practices. Pearson Education, Inc.

[Met08] Methods & Tools 2008. Adoption of Agile Methods poll. <http://www.methodsandtools.com/dynpoll/oldpoll.php?Agile2> 2008

[Mou08] Mountain Goat Software 2008. The Scrum development process. <http://www.mountaingoatsoftware.com/scrum>

[Nebu05] Nebulon Pty., Ltd. 2005. Feature Driven Development overview. <http://www.nebulon.com/articles/fdd/download/fddoverview.pdf>.

[Osh06] Oshana, Robert 2006. DSP Software Development Techniques for Embedded and Real-Time Systems. Elsevier Inc.

[Palm02] Palmer, Stephen R.; Felsing, John M. 2002.A Practical Guide to Feature-Driven Development. Prentice Hall

[Poppen03] Poppendieck, Mary; Poppendieck, Tom 2003. Lean Software Development: An Agile Toolkit for Software Development Managers. Addison-Wesley

[Poppen03b] Poppendieck, Tom 2003. Agile Customer's Toolkit. <http://www.poppendieck.com/pdfs/Agile_Customers_Toolkit_Paper.pdf >

[Poppen07] Poppendieck, Mary; Poppendieck, Tom 2007. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley

[Roy07] Royce, Winston W. 1970. Managing the Development of Large Software Systems: Concepts and Techniques. Technical Papers of Western Electronic Show and Convention. August 25-28, 1970.

[Scho04] Van Schooenderwoert, Nancy 2004. Embedded Extreme Programming: An Experience Report. Presented in ESC conference in Boston 2004. <http://www.agilerules.com/articles/Embedded_Extreme_Programming_Experience_Report.pdf>

[Schw01] Schwaber, Ken; Beedle, Mike 2001. Agile Software Development with Scrum. Prentice Hall

[Schw06] Schwaber, Ken 2006. Introduction to Scrum <http://www.controlchaos.com/resources/intro.php>

[Schw07] Schwaber, Ken 2007. The Enterprise and Scrum. Microsoft Press

[Shi03] Shine Technologies 2003. Agile Methods Survey Results. 2003

[Sim99] Simon, David E. 1999. An embedded software primer. Addison-Wesley

[Som04] Sommerville, Ian 2004. Software Engineering, Seventh Edition. Pearson Education Limited

[Sta94] Standish Group International, Inc. 1994. CHAOS Report. <http://www.standishgroup.com/sample_research/chaos_1994_1.php>.

[Sta04] Standish Group International, Inc. 2004. CHAOS Report: CHAOS Chronicles.

[Step03] Stephens, Matt; Rosenberg, Doug 2003. Extreme Programming Refactored: The Case Against XP. Apress

[Stew99a] Stewart, David B. 30 Pitfalls for Real-Time Software Developers, Part 1. Embedded Systems Programming Magazine, vol.12, no 11, p. 32-41. October 1999. <http://www.ece.umd.edu/serts/bib/mags/esp99a.pdf>

[Stew99b] Stewart, David B. More Pitfalls for Real-Time Software Developers. Embedded Systems Programming Magazine, vol.12, no 12, p. 74-86. November 1999. <http://www.ece.umd.edu/serts/bib/mags/esp99b.pdf>

[Sub06] Subramaniam, Venkat; Hunt, Andrew 2006. Practices of an Agile Developer: Working in the World. The Pragmatic Bookshelf

[Tak86] Takeuchi, Hirotaka; Nonaka, Ikujiro 1986. The New New Product Development Game. Harvard Business Review, January-February 1986. 137-146

[Tak95] Takeuchi, Hirotaka; Nonaka, Ikujiro 1995. The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation. Oxford University Press

[Ten03] Tennies, Nathan 2003. Software Matters for Power Consumption, Embedded Systems Programming, February 2003.

[Ver07] Version One: 2nd Annual State of Agile Survey, conducted June-July 2007. 2007

[Ver08] Version One: 3rd Annual State of Agile Survey, conducted June-July 2008. 2008

[Ver09] Version One: 4th Annual State of Agile Survey, conducted June-November 2009. 2009

[Vil08] Vilkki, Kati 2008. Juggling with the Paradoxes of Agile Transformation. Keynote at XP2008 conference in Limerick.

[Xil00] Nicklin, Dave 2000. Xilinx at Work in Set-Top Boxes. <http://direct.xilinx.com/bvdocs/whitepapers/wp100.pdf>

[Yag07] Yaghmour, Karim 2003. Building Embedded Linux Systems. O'Reilly