Requirements Engineering in Agile Software Development

Diploma Thesis

Frauke Paetsch, 97429

Fachhochschule Mannheim
Informationstechnik
Dr. Eckhard Koerner

University of Calgary
Department of Electrical & Computer Engineering
Dr. Armin Eberlein

Department of Computer Science
Dr. Frank Maurer

March 31st, 2003

Ehrenwörtliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

Calgary, den 31.03.2003

Frauke Paetsch

German Abstract

Das Thema der Diplomarbeit ist 'Requirements Engineering in Agile Software Development'. Der erste Teil der Diplomarbeit befasst sich mit den beiden Software Engineering Ansätzen und vergleicht diese. Während das traditionelle Requirements Engineering mehr dokumentenorientiert ist, versuchen agile Methoden Dokumentation so weit wie möglich zu reduzieren. Die Diplomarbeit analysiert die Unterschiede und Gemeinsamkeiten der beiden Ansätze und zeigt, wie agile Methoden von Requirements Engineering Techniken profitieren könnten. Der zweite Teil der Diplomarbeit zeigt, wie eine der untersuchten Requirements Engineering Techniken in einem Tool, das verteilte Softwareentwicklung unter Anwendung agiler Methoden unterstützt, integriert werden kann.

Abstract

The topic of this paper is requirements engineering (RE) in agile software development. The first part concentrates on the comparison of traditional requirements engineering approaches and agile software development. While traditional requirements engineering methods are often document-centric, agile methods often try to minimize documentation. We analyze commonalities and differences of both approaches and determine how agile software development can benefit from traditional engineering methods. The second part shows how one of these RE techniques can be integrated in a tool that supports distributed agile software development over the internet.

Contents

1 Introduction

Agile software development approaches have become increasingly popular during the last few years. Agile practices have been developed with the aim to deliver software faster and to ensure that the software meets changing needs of customers. In general, these approaches share some common principles: improving customer satisfaction, adapting to changing requirements, frequently delivering working software, and close collaboration of the customer and developers. Agile approaches are based on short iterations presenting a new version of the software every one to three months.

Requirements engineering (RE) is a traditional software engineering process whose goal it is to identify, analyze, document and validate requirements for the system to be developed. Often, doing requirements engineering and practicing agile approaches is seen to be incompatible: RE often relies on documentation for knowledge sharing while agile methods focus on face-to-face collaboration between customers and developers to reach similar goals. The aim of this paper is to determine how requirements engineering techniques are used within agile development and if agile approaches could be improved by the use of certain RE techniques.

RE is a subprocess in the traditional software development process. Considering the waterfall model, a standard model in software engineering, RE is the first of five phases. The five phases in the waterfall model are: requirements definition, system and software design, implementation and unit testing, integration and system testing, and operation and maintenance. These five phases are executed successively and result in a functioning software product.

But the software development process has experienced changes during the last years. One of the changes are the agile software development approaches. The internet also has had a significant impact on the way software is developed. People are not always working together in the same room, building, city, or even country and have to co-ordinate their work for example over internet. There exist several tools which support distributed development. One of this

tools is M-ASE (MILOS Agile Software Engineering). It was specially developed for distributed software development using agile approaches and supports coordinated planning and management of iterations and tasks.

Most of the agile approaches include prioritization to ensure that the delivered software is most valuable for the customer. Prioritization is used to rank the features (or tasks) to be implemented in order of their importance and features with the highest priority will be implemented first. In RE, prioritization is used to enable developers to focus on the most important features if there is inadequate time for implementation. As prioritization is such an important technique both in agile development and in RE we decided to add a functionality to M-ASE which will allow groups of users to prioritize the tasks per iteration.

The next section gives an overview on current requirements engineering techniques. Section 3 discusses agile approaches from a requirements engineering perspective. In Section 4, we evaluate how the incorporation of some requirements engineering techniques could improve agile methods. Section 5 describes the objectives of developing a prioritization tool for the M-ASE System and the M-ASE System itself. Section 6 contains the development of the prioritization feature. The last section is a short summary of the paper.

2 Requirements Engineering (RE)

2.1 Overview

Requirements Engineering is concerned with identifying, modeling, communicating and documenting the requirements for a system, and the contexts in which the system will be used. Requirements describe what is to be done but not how they are implemented [Dav94]. There are many techniques available for use during the RE process to ensure that the requirements are complete, consistent and relevant. The aim of RE in the waterfall model is to help to know what to build *before* system development starts in order to prevent costly rework. This goal is based on two major assumptions:

- The later mistakes are discovered the more expensive it will be to correct them [KS97].
- It is possible to determine a stable set of requirements before system design and implementation starts.

In the following, we will briefly examine the requirement engineering process and discuss the main techniques that were developed for it.

2.2 Requirements Engineering process

The RE process consists of five main activities [KS97]:

- Elicitation
- Analysis and Negotiation
- Documentation
- Validation
- Management

They can be presented in different models. The most popular models are:

1. The Coarse-Grain Activity Model (see Figure 1). The cloud icons indicate that there are no distinct boundaries between the activities.

2. The Waterfall Model. Showing a sequence of phases following each other.

3. The Spiral Model (see Figure 2). It indicates that the different activities are repeated until a decision is made that the requirements document should be accepted.

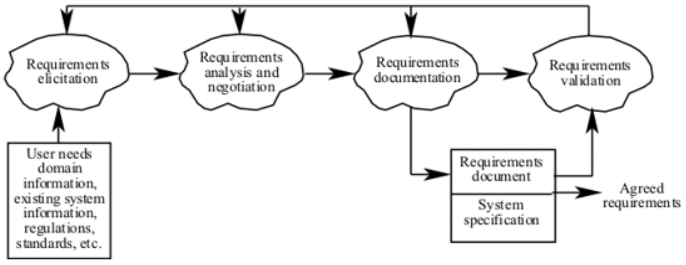

Figure 1: The Coarse-Grain Activity Model [KS97]

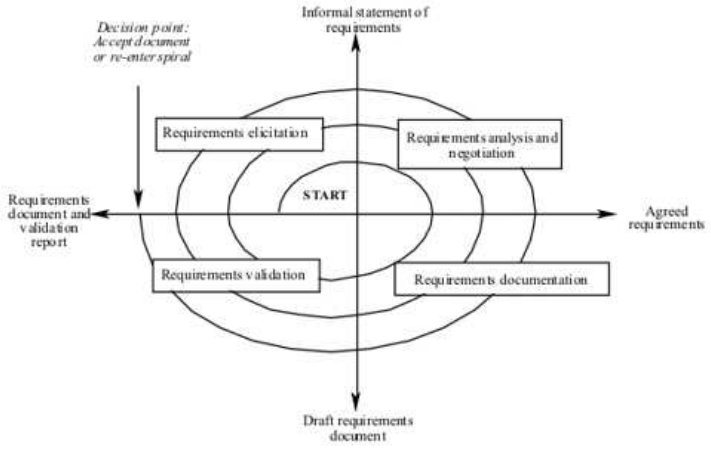

Figure 2: The Spiral Model [KS97]

The Inputs and Outputs of the RE process are shown in Figure 3. Although the RE process varies form one organization to another, the inputs and outputs are similar in most cases. RE helps to detect mistakes earlier, which reduces costs of software development. The later mistakes are discovered the more expensive it will be to correct them [KS97].

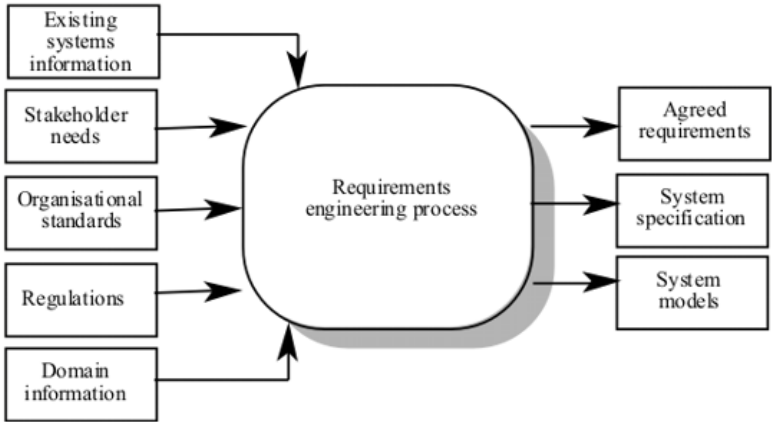

Figure 3: RE process - Inputs and Outputs [KS97]

2.3 Requirements Elicitation

2.3.1 Overview

Elicitation refers to the identification of the requirements and system boundaries through consultation with stakeholders (e.g., clients, developers, users). The system boundaries affect all elicitation techniques and define the context of the delivered system. Full Understanding of the different areas (e.g., application domain, business needs and constraints of system stakeholders, the problem itself) is essential to envision the system that is to be developed. The most important techniques for requirements elicitation are described in the remainder of this section.

2.3.2 Interviews

Interviewing is a method for discovering facts and opinions held by potential users and other stakeholders of the system under development. Mistakes and misunderstandings can be identified and cleared up. There are two different kinds of interviews:

- the closed interview, where the requirements engineer has a pre-defined set of questions and is looking for answers
- the open interview, without any pre-defined questions the requirements engineer and stakeholders discuss in an open-ended way what they expect from a system.

In fact, there is often no distinct boundary between both kinds of interviews. You start with some questions which are discussed and this leads to new questions [KS97]. The advantage of interviews is that they help the developer to get a rich collection of information. Their disadvantage is that this amount of qualitative data can be difficult to analyze and different stakeholders my provide conflicting information.

2.3.3 Use cases / Scenarios

Use cases describe interactions between users and the system, focusing on what users need to do with the system by identifying the most important functions. A use case specifies a sequence of interactions between a system and an external actor (e.g., a person, a piece of hardware, another software product), including variants and extensions, that the system can perform. The use case provides the result to a specific task demanded by one actor. Use cases represent requirements of the software system and can be used during the early stages in the development process. Analysts and customers should examine every proposed use case to validate it [E2S03], [Wie99].

Scenarios are examples of interaction sessions where a single type of interaction between user and system is simulated. Scenarios should include a

description of the state of the system before entering and after completion of the scenario, what activities might be simultaneous, the normal flow of events and exceptions to the events [KS97].

2.3.4 Brainstorming

Brainstorming is a possibility to develop creative solutions related to a certain topic. Normally, brainstorming is a group activity but you can also brainstorm on your own. Brainstorming contains two phases - the generation phase, where ideas are collected, and the evaluation phase, where the collected ideas are discussed. In the generation phase, the ideas shouldn't be criticized or evaluated. The ideas should be developed fast and be broad and odd. Every idea can lead to new ideas. Brainstorming leads to a better understanding of the problem for everyone and a feeling of common ownership of the result.

2.3.5 Observation and social analysis

Observational methods involve an investigator viewing users as they work in a field study, and taking notes on the activity that takes place. Observation may be either direct with the investigator being present during the task, or indirect, where the task is viewed by some other means such as through use of a video recorder. The method is useful early in user requirements specification for obtaining qualitative data. It is also useful for studying currently executed tasks and processes. Observation allows the observer to view what users actually do in context - overcoming issues with stakeholders describing idealized or oversimplified work processes. Direct observation allows the investigator focused attention on specific areas of interest. Indirect observation records activities and allows the detection of issues that might have gone unnoticed in direct observation.

2.3.6 Focus Groups

Focus groups are an informal technique where a group of four to nine users from different backgrounds and with different skills discuss in a free form issues and concerns about features or a prototype of a system. Focus groups help to

identify user needs and feelings, what they think, what things are important to them and what they want from the system. They often bring out spontaneous reactions and ideas. The moderator should follow a pre-planned script with different topics, which are introduced one at a time to find out what people think of the topic. For the participants the session should feel free-flowing. Since there is often a major discrepancy between what people say and what they do, observation and social analysis should additionally be involved. Focus groups can support the articulation of visions, design proposals and a product concept. Additionally, they help users in analysing their own problems and things that should be changed, and support the development of a 'shared meaning' of the system [Mac96], [Nie97].

2.3.7 Soft System Methodology (SSM)

Soft System Methodology focuses on a desirable system and how to reach it. It supports analysis of a problem situation from a number of different perspectives. SSM can be used when there is a situation in everyday life with which at least one person sees problems and wants improvements. SSM looks at the human and organisational context within which the desired system will operate, rather than just at the system itself. The conventional SSM model consists of seven fundamental stages [Mac96], [CS99]:

1. Description of problem situation

   Who is involved and what is their view of the situation? What is the organisational context?

2. Picture of the world

   The problem situation is expressed in a way which helps choosing a relevant system in stage 3.

3. Root definition of relevant system

   - Define imaginary systems being relevant to the problem situation

- Guideline for well-formulated Root Definition is CATWOE[Mac96]

  C: Customers benefiting of the systems

  A: Actors carrying out the defined activities

  T: Transformations of inputs and outputs

  W: Weltanschauung (view of the world)

  O: Owner having power to authorize or dismantle system

  E: Environmental Constraints

4. Conceptual Model

   A human activity model showing dependencies between activities, which exactly matches a root definition.

5. Comparison of real world a model

   Activities are compared with what happens in the real world. Special questions are asked for each activity:

   - Does the activity take place in the real world?
   - How is the activity done?
   - How can performance be measured?
   - Is the activity processed effectively?

6. Identifying feasible and desirable changes

   Which activities are culturally feasible and systematically desirable?

7. Action to improve problem situation.

2.3.8 Requirements reuse

Requirements reuse means reusing of requirements specifications or parts thereof developed for former projects. Requirements reuse can be compared with using of templates and be simple when the problem of two projects is quite similar [McP01]. Requirements reuse can reduce the general effort inside it [LL01]. Requirements reuse is only possible in certain cases:

- The requirement contains information about application domain - Requirements normally don't specify a specific system functionality but constraints on the system or system operations which are derived from the application domain.
- The requirement is concerned with the style of presentation - Having a certain "look and feel" for software products developed for a specific company makes sense.
- The requirement reflects company policies (security policies).

2.3.9 Prototyping

A prototype of a system is an initial version of the system which is available early in the development process. Prototypes of software systems are often used to help elicit and validate system requirements. There are two different types of prototypes:

Throw-away prototype It helps to elicit requirements which cause difficulties in understanding.

Evolutionary prototype It delivers a workable system to the customer and can become a part of the final system.

These prototypes can be implemented as a paper prototype, where a mock-up of the system is developed and used for system experiments, as a "Wizard of Oz" prototype, where a person simulates the responses of the system in response to some user inputs or as an automated prototype, where a fourth generation language or some other rapid development environment is used to develop an executable prototype.

2.4 Requirements Analysis and Negotiation

2.4.1 Overview

Requirements Analysis checks requirements for necessity (the need for the requirement), consistency (requirements should not be contradictory) and com-

pleteness (no service or constraint should be missing), and feasibility (requirements are feasible in the context of the budget and schedule available for the system development). Conflicts in requirements are resolved through prioritization negotiation. Requirements that seem problematic are discussed and the stakeholders involved present their views about the requirements. Disputed requirements are prioritised to identify critical requirements and to help the decision making process. Solutions to requirements problems are identified and a compromise set of requirements is agreed upon. Generally, this will involve making changes to some of the requirements. The main techniques used for requirements analysis are JAD sessions, Prioritization, and Modeling.

2.4.2 Joint Application Development (JAD)

JAD was developed in 1977 by IBM and first used in the 1980s. JAD is a facilitated group session (or workshop) with a structured analysis approach. The purpose of JAD is to define a special project on various levels of details, to design a solution, and to monitor the project until it's completed. Participants include executives, project managers, users, system experts and external technical personnel [Mac96]. There are different roles, which should be represented at each group session:

- A Session leader
- A User representative
- A specialist
- An analyst
- An information system representative
- An executive sponsor

The session leader is responsible for the course of the process. S/he facilitates debate and preparation of documents [Mac96].

2.4.3 Requirements Prioritization

On a project with a tight schedule, limited resources, and high customer expectations it is essential to deliver the most valuable features as early as possible. When the date for shipping is close but not all features are implemented some features have to be discarded. Setting priorities early in the project helps to decide which features to skip. Requirements Prioritization should be done by the customer. If a third party makes this decision for the customer s/he might disagree on the prioritization. Both customer and developer have to provide input to requirements prioritization. The customer marks those features with the highest priority providing the greatest benefit to users. Developers point out the technical risks, costs, or difficulties. With this background the customer might change priority of some features. Developers also might propose to implement a feature having a higher impact on the system's architecture but with lower priority earlier [Wie99]. Prioritization is common practice in requirements engineering. Various prioritization techniques (like pair-wise comparison or the Analytic Hierarchy Process (AHP)) have been used in requirements engineering for numerous years.

2.4.4 Modeling

System models are an important bridge between the analysis and the design process. A number of methods use different modelling techniques to formulate or analyze system requirements. The most popular modelling techniques are [KS97]: data-flow models, semantic data models and object-oriented approaches.

Data-flow modelling The data-flow model is based on the notation that systems can be modelled as a set of interacting functions. It uses data-flow diagrams (DFDs) to graphically represent the external entities, processes, data-flow, and data stores. (See figure 4 for the data-flow diagram notation)

Semantic data model A system model should also describe the logical form

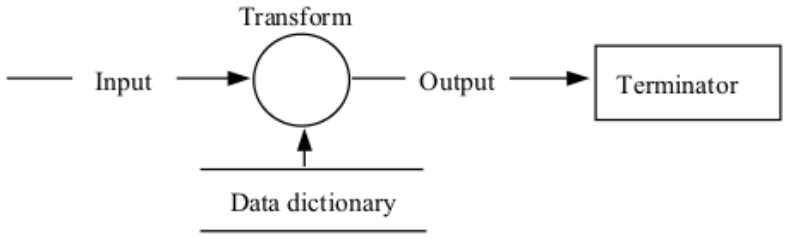

Figure 4: Data-flow diagram notation [KS97]

data is processed by the system. This can be done using the relational model in which data is specified as a set of tables with some columns representing common keys regardless the physical organization of the database. The semantic data model includes the entity-relationship model. The entity in a database, the attributes belonging to it and explicit relationships between them are identified. (Figure 5 shows the notation for semantic data models)

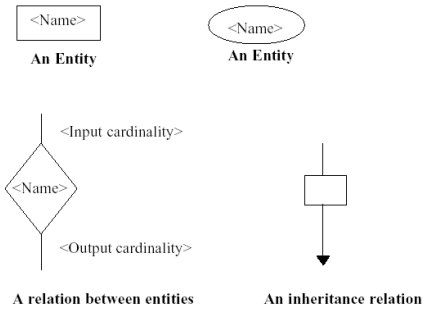

Figure 5: Notation for semantic data models [KS97]

Object-oriented approaches The basic principle of an object-oriented model is the notion of an object. An object defines a set of common attributes and operations associated with it (see figure 6). The basic concepts for object-oriented approaches are:

- class
- operations of services
- encapsulation
- inheritance

Communication between objects is based on messages causing an operation to be invoked. The operation performs the appropriate method and returns a response.

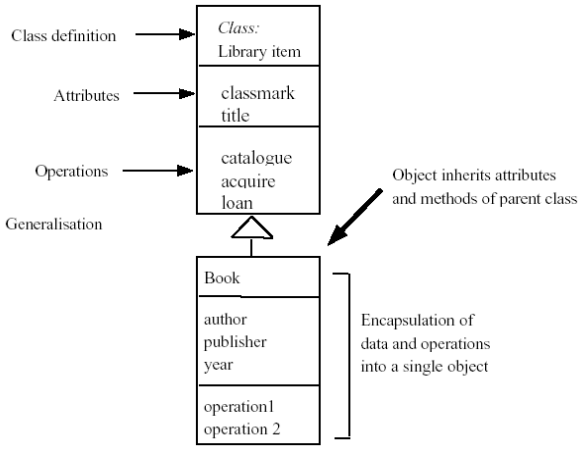

Figure 6: An illustration of object-oriented concepts [KS97]

2.4.5 Quality Function Deployment (QFD)

QFD was developed by the Japanese automobile industry. The goal was to design customer satisfaction into a product before it was manufactured by translating customer requirements into engineering specifications. QFD includes group sessions in which the 'House of Quality' is used to focus attention. The 'House of Quality' (see figure 7) surrounds a Matrix showing the relationship between the customer requirement and the proposed feature. The triangular "roof" matrix of the 'House of Quality' is used to identify where the technical requirements that characterize the product support or impede one another.

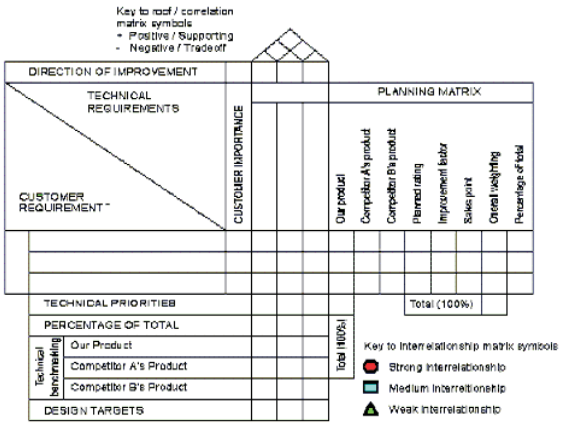

Figure 7: House of Quality

2.5 Requirements Documentation

The purpose of requirements documentation is to communicate the understanding of the requirements between stakeholders and developers. The requirements document explains the application domain and the system to be developed. It can be the baseline for evaluating subsequent products and processes (system design, testing cases, verification and validation activities) and for change control. A good requirements document is clear/unambiguous, complete, correct, understandable, consistent, concise, and feasible. Depending on the customer-supplier relationship, the specification document can be part of the contract [Mac96].

2.6 Requirements Validation

2.6.1 Overview

The purpose of Requirements Validation is to certify that the requirements represent an acceptable description of the system to be implemented. Inputs for the validation process are the requirements document, organisational standards, and organisational knowledge. The Outputs are a problem list that contains the reported problems with the requirements document and the agreed actions, to

cope with the reported problems. Techniques used for requirements validation are Requirements Review and Requirements Testing. Requirements Validation should usually result in sing-offs from all project stakeholders [KS97].

2.6.2 Requirements Reviews

Requirements Reviews will ensure that requirements:

- Are complete
- Are consistent
- Are unambiguous
- Can be tested

There are informal and formal requirements reviews. The purpose of informal requirements reviews is to criticize and improve the requirements. The informal requirement review takes place incrementally and throughout the RE process with interested stakeholders. The purpose of formal requirements review is to approve the requirements document and to authorize the project. Therefore it takes place when the requirements document is complete [Pot98].

2.6.3 Requirements Testing

Testing Software is an integral part of software engineering. But if software is written based on misunderstood requirements the result will not be adequate. For this reason writing tests for requirements should be started as soon as work on the requirements for a product is started [Rob00]. The test written during requirements analysis can be executed in the final system to validate the requirements. The benefit of writing tests is that problems with requirements are often discovered before design and implementation [McP01].

2.6.4 Prototyping

Throw-away prototyping can be used to determine the feasibility of a particular approach to implementation of a system.

2.7 Requirements Management

Requirements management includes all activities concerned with Change Control, Version Control, Requirements Tracing, and Requirements Status Tracking (see figure 8). The goal of requirements management is to capture, store, disseminate, and manage information. Management of information means organization, analysis and traceability. Traceability is a technique used to provide relationships between requirements, design, and implementation of a system in order to manage changes to a system [Wie99].

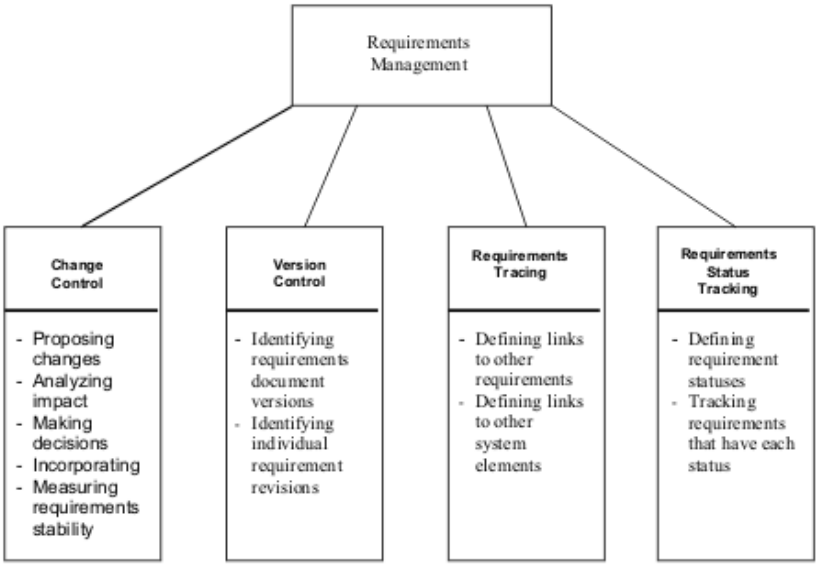

Figure 8: Requirements Management Activities [Wie99]

3 Agile Development

3.1 Overview

In comparison to traditional software development agile development is less document-oriented and more code-oriented. This, however, is not its key characteristic but rather a reflection of two deeper differences between the two styles [Fow00]:

- Agile methods are adaptive rather than predictive. With traditional methods most of the software process is planned in detail for a longer time period. This works well if not much is changing (i.e. low requirements churn) and the application domain and software technologies are well understood by the development team. Agile methods were developed to adapt and thrive on changes.
- Agile methods are people-oriented rather than process-oriented. They rely on people's expertise, competency and direct collaboration rather than on rigorous, document-centric processes to produce high-quality software.

In this section, the most common agile methods are briefly discussed.

3.2 Extreme Programming (XP)

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are.

The XP Process

The life cycle of an ideal XP project consists of six phases: Exploration, Planning, Iterations to first Release, Productionizing, Maintenance and Death.
In the *Exploration Phase* customers write stories describing the system functionality. Each story describes a feature of the system. At the same time the

programmers explore every piece of technology they will be using. This phase might take a few months if the team has to use completely new technology or work in a new domain.

In the *Planning Phase* customers prioritize the stories and programmers estimate the effort for each story. Both agree on a date for the first (small) release and on a set of valuable stories for this release. The planning phase itself should take a couple of days.

The result of the last iteration of the *Iteration to first Release Phase* is the first release, and the product is ready for production. The programmers choose stories for each one- to four-week iteration. During each iteration functional tests for the chosen stories are produced and run at the end of the iteration. By selecting stories for the first iteration which enforce the structuring of the whole system, the architecture is put in place within this iteration.

During the *Productionizing Phase* it might be necessary to shorten the iterations from three weeks to one week. Performance testing and a change of stories may also be needed. Changes to stories are only included in the current release after discussion with the customer. New tests are added to prove the quality of the system.

The *Maintenance Phase* is the normal state of an XP project. New features are developed, the system is kept running, and new people may be incorporated into the team when the team structure is changed to deal with production. Developing a system in production is done more carefully.

The *Death Phase* is near if customers are unable to generate a new story. This is the time where the necessary documentation is written. Another reason for the Death of a project is when it becomes too expensive or if it doesn't deliver the desired outcomes.

XP Practices

Many of the XP practices are old, tried and tested but often forgotten. XP resurrects and combines these techniques in such a way that they reinforce each

other.

- The Planning Game

  Quickly determine the scope of the next release/iteration by combining business priorities and technical estimates. As reality overtakes the plan, update it.

- Frequent small releases

  Put a simple system into production quickly, then release new versions on a very short cycle.

- On-site customer

  Include a real, live customer on the team to answer questions.

- Testing

  Programmers continually write unit tests before they write production code; customers write tests demonstrating that features are finished.

- Simple Design

  The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.

- Refactoring

  Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify or add flexibility.

- Pair programming

  All production code is written with two programmers at one machine.

- Collective code ownership

  Anyone can change any code anywhere in the system at any time.

- Continuous integration

  Integrate and build the system many times a day, every time a task is completed.

- Sustainable development

  Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.

- Coding standards

  Programmers write all code in accordance with rules emphasizing communication through the code.

- Metaphor

  Guide all development with a simple shared story of how the whole system works.

Following is a short overview of how these practises are used within XP.

A team for XP-projects should work together in one room, where the workstations (max. number is number of programmers divided by two) are clustered in the middle of the room. An iteration-cycle takes three weeks; at the end of each iteration running, tested code can be presented to the customer. A release period could last two to five iterations. The project-metaphor helps everyone to understand the basic elements of the project and their relationships. Instead of an SRS or another requirements document, XP relies on "user-stories", user-visible functionalities which can be implemented during one iteration. These user-stories are written on index-cards, and discussed with the customer when development on a story starts. For this reason, the text on the card can be very short. The conversation leads to an understanding of the requirements. During the "Planning Game" for each release, customers write the user-stories on the cards. The developers estimate how long it will take to implement each story. If they are not able to estimate a certain story, the story can be split. The customers and developers sort the cards by value and risk and choose the ones for the next release.

The system will be developed using Pair programming and following a strict coding standard. Programmers are to change the code when they find an easier/clearer way to do something (refactoring). This is possible because of the coding standard and the code belonging to everyone in the team (collective

code ownership). Refactoring keeps the code simple. Before coding, programmers develop unit tests for everything they code, which must run at 100% before integration. Integration interval is from 15 minutes to a few hours. During development, customers can think about new user-stories, tests and talk with the programmers to clarify ideas. They can be on-site full-time but this may be deemed unnecessary [Bec99], [Coc02].

XP does not explicitly refer to requirements techniques in detail but about the general software development process and what is to be done during the process. Several XP practices (or techniques used in this practices) can be compared with lightly modified RE techniques. Specifically during the Planning Game, RE elicitation techniques like interview, brainstorming and prioritization are used. The main 'tool' used for elicitation are the story cards on which users write their user stories. We want to highlight that user stories are different from use cases. A user story is a description of a feature that provides business value to the customer. Use cases are a description of interactions of the system and its users(for example a special dialog) and do not necessarily provide business value.

Before the stories can be written on the cards customers have to think about what they expect the system do to and what functionality is needed. This process is a kind of brainstorming. Thinking about a special functionality leads to more ideas and to more user stories. Every story is discussed in an open-ended manner prior to implementation. Developers ask for more details to determine what customers really want the system to do and estimate the effort required to implement the story. Based on these estimates and the time available in the next iteration, customers are in turn able to choose stories to be developed. The stories selected should contain the most important features for the customers.

XP emphasizes writing tests before coding. Not only do developers write unit tests to ensure the code is working, but customers also write acceptance tests for each story. When customers write tests they ask themselves what should be checked before they would accept the story to be done. Tests are run every time a new piece of code is to be integrated into the system.

XP is based on frequent small releases which can be compared with requirements review and with evolutionary prototyping. The difference between XP and prototyping is that XP requires a tested, cleanly designed system while prototypes can be ill-designed and 'hacked together'. Every release only contains the next set of most valuable business requirements. Customers can review and test the functionality and design of the delivered program and discuss issues they want to be changed or added in the next release.

3.3 Agile Modeling (AM)

The key focus of Agile Modeling is a set of core and supplementary principles and modeling practices. The core principles, supported by the supplementary principles, of AM focus on the importance of producing usable software. Like XP, AM points out that changes in software development are normal and have to affect the way of development, i.e. the way of modeling.
AM's basic values are the same as with XP: communication, simplicity, feedback, and courage. Additionally AM refers to humility pointing out that people with different backgrounds are working together[Amb01].

Following is a description of the principles and practices[Amb01]:

Principles:

- Assume Simplicity
  The simplest solution is the best solution. Just build what you need today and believe that you'll be able to build tomorrow what you need tomorrow.
- Embrace Changes
  Requirements, people's understanding of requirements, and project stakeholders can change over time. The project should reflect these changes.
- Enabling the next effort is your secondary goal
  To be able to develop the next major release, enough documentation and supporting material should be generated.

- Incremental Change

  You can't do everything right the first time, just make it good enough at the time and evolve it over time.

- Maximize Stakeholder Investment

  Stakeholders should be able to decide how their invested resources are used/spent.

- Model with a purpose

  When modeling, always model for someone/something (customer/communication). If you're not able to think of a purpose then first rethink if you really need the model.

- Multiple models

  Always try different models, because every model has a special "focus".

- Quality Work

  "Good" work is easier to understand and something people can be proud of.

- Rapid Feedback

  The faster you can get feedback for your work/answers to your questions, the more you can learn and the better your work will be.

- Software is your primary goal

  The goal is not to produce lots of documentation, models, etc. but software that meets the customers needs.

- Travel light

  Just keep the models needed. The less models, the less to update.

- Content is more important than representation

- Everyone can learn from everyone else

  Working with others is a good opportunity to learn and to try new ways of doing something.

- Know your models

  Every model has strengths and weakness which should be known.

- Know your tools

  Understand the features of the tools you are working with.

- Local Adaption

  When AM is used it has to be adapted to the environment and to the nature of the organization.

- Open and honest communication

  People should never be afraid to state their opinion or to share ideas. This increases quality of information leading to better decisions.

- Work with people's instincts

  When people have a special feeling about something, a decision based on this feeling is the right one in most cases.

Practices:

- Active stakeholder participation

  Additional to the on-site customer, AM asks for actively involving stakeholders (direct users, management). This means, for example, timely resourcing decisions by management.

- Apply the right artifact(s)

  Every artifact has a specific application where it's the most valuable. Sometimes a diagram is worth more than 1000 lines of code or 10 pages of documentation.

- Collective ownership

  Everyone can work on everything belonging to the project at any time.

- Consider testability

  Always ask yourself how you are going to test what you are modeling. If you can't think of a test, you should stop working on that particular piece.

- Create several models in parallel

  Because of the different strengths and weaknesses of the models a single model may not be enough.

- Create simple content

  Keep models as simple as possible but valuable.

- Depict models simply

  A very simple model (like a class model with the primary responsibilities) can be more useful than a complicated one.

- Display models publicly

  Displaying models on a "modeling wall" supports "open and honest communication" and all current models are quickly accessible. Stakeholders see that there is nothing to hide from them.

- Iterate to another artifact

  When working on a model and don't know how to go on, try another model.

- Model in small increments

- Model with others

  You often model to communicate or to understand. This is a process valuable for the whole team and input from several people is more effective.

- Prove it with code

  To find out if an idea will work code it and find out!

- Use the simplest tools

  Most models can be drawn on whiteboards or paper. When you have to save them, take a picture, but mostly they are throwaways.

- Apply modeling standards

  Developers should agree and follow a set of modeling standards.

- Apply patterns gently

- Discard temporary models

  Most models are temporary, so throw them away when they fulfilled their purpose.

- Formalize contract models

  When working with external groups keeping required system information (database, legacy application) both parties should agree to a contract model and change it over time, when necessary.

- Model to communicate

  Models encourage and support communication.

- Model to understand

  To understand a certain problem, identify or analyze requirements.

- Reuse existing resources

- Update only when it hurts

  Models don't have to be perfect to be valuable. Only update when updating provides more value.

AM does not explicitly refer to any RE techniques but some of the practices of AM support several RE techniques (e.g. tests and brainstorming). The basic idea of AM is to give developers a guideline of how to build models that resolve design problems but not 'over-build' the models. AM highlights the difference between informal models whose sole purpose is to support face-to-face communication and models that are preserved and maintained as part of the system documentation. The latter are what is often found in RE approaches.

3.4 Scrum

Scrum is a method for managing the system development process by applying some ideas of industrial process control theory (flexibility, adaptability and productivity) [SB01]. Scrum does not propose any particular software development technique for implementation. It focuses on how a team should work together

to produce quality work in a changing environment[PAW02].

The Scrum Process

The phases of the Scrum Process are: pre-game, development and post-game[PAW02].
The *pre-game phase* consists of two sub-phases. During the `planning phase` the definition of the system is developed. This includes the Product Backlog, a prioritized list changing continuously and identifying all outstanding work. The requirements (coming from customers, sales and marketing division, customer support or software developers) are prioritized and time for their implementation is estimated. The `architecture phase` contains the high level design of the system and system architecture.
In the *development phase* the system is developed. The development process is divided into Sprints. Sprints are work increments (lasting 30 days) in which the team completes the prioritized tasks chosen for this sprint. Every day there is a 15 min meeting called Scrum, where work already done is discussed, outstanding work is identified, and recognized obstacles are discussed. A Sprint itself consists of traditional software development phases: requirements, analysis, design, evolution, delivery[PAW02]. Before each sprint, functionality required for the sprint is defined. The goal is to have stable requirements during the sprint[SB01].
The *post-game phase* begins when customers agree that requirements are completed. The system can be integrated and the necessary documentation can be written.

The main Scrum techniques are the Product Backlog, Sprints, and Scrums. With regard to Requirements Engineering the product backlog plays a special role in Scrum. All the requirements regarded as necessary or useful for the product are listed in the product backlog. It contains a sorted list of all features, functions, enhancements, and bugs. The product backlog can be compared with an incomplete and changing (a kind of: living) requirements document containing information needed for development. Each sprint, a 30 day development iteration, is planned based on the information included in the product backlog.

Selected tasks from the product backlog are moved to the sprint backlog. No changes are made to the sprint backlog during the sprint. I.e. there is no flexibility in the requirements during a sprint but there is absolute flexibility for the customer picking the requirements for the next sprint.

During a Sprint the development team goes through several stages (Sprint Planning Meeting, Sprint, Sprint Review). The goal of the *sprint review meeting* is that the participants (customers and developers) gain understanding of the system, its technical architecture and design as well as the delivered functionality. The knowledge gathered in the sprint review meeting and the current product backlog is the basis for the next sprint planning meeting. A part of the sprint review meeting, where the participants gain understanding of the system, can be compared to requirements review and a presentation of an evolutionary prototype to the customer.

3.5 The Crystal Methodologies

The Crystal Methodologies are a family of different methodologies from which the appropriate methodologies can be chosen for each project. The different members of the family can be tailored to fit varying circumstances.
The members of the Crystal family are indexed by different colours to indicate the "heaviness": Clear, Yellow, Orange, Red, Magenta, Blue, Violet, and so on. The darker the colour, the "heavier" the methodology. A project with 80 people needs heavier methodologies than one with only 10 people[PAW02].

The Crystal Family Members share two values[Coc02]:

- Tolerance (every tool, process or work product can be used)
- Communication and cooperation of people

two rules:

- Incremental development (one to four (better three) months)
- Pre- and post-increment reflection workshops

and two base techniques:

- The methodology-tuning technique (convert a base methodology to a starter methodology)
- The technique for the reflection workshops

All of the Crystal Family Members yet known have several "methods" in common. They provide guidelines for *policy standards* like incremental delivery, direct user involvement or automated testing. Several *work products* (common object models, user manual) are suggested. Some basic *tools* for versioning, programming, testing, communication, project tracking, drawing, and performance measuring are required. Crystal also recommends the use of notation *standards*, design conventions, formatting standards and quality standards [PAW02].

Up to now three Crystal methodologies have been constructed and been used. These are Clear, Orange, and Orange Web. The difference between Orange and Orange Web is that Orange Web does not deal with a single project. As there is not much literature about Crystal, the following policy standards of Clear and Orange can be compared with RE techniques but will not be consider in analysis [PAW02]:

- Incremental delivery on a regular basis (Prototyping, Reviews)
- Automated regression testing of functionality (Testing)
- Two user viewings per release (Review)
- Workshops for product- and methodology-tuning at the beginning and in the middle of each increment (Review)

3.6 Feature Driven Development (FDD)

FDD is a short-iteration process for software development focusing on the design and building phase [PAW02]. FDD does not recommend a specific process model. FDD iterations last two weeks [Fow00].

FDD consist of five sequential processes. The first three are done at the beginning of the project and the last two during each iteration[Fow00].

- Develop an Overall Model
- Build a Features List
- Plan by Feature
- Design by Feature
- Build by Feature

The two most important roles in FDD are the *Chief Programmer*, an experienced developer leading small development teams and participating in the requirements analysis and design of the project, and the *Class Owner*, working on the class being assigned [PCL99].

In the first phase an overall model is developed by domain members and developers. The feature list is built in the second phase and based on the overall model. The overall model consists of class diagrams with classes, links, methods, and attributes. The classes and links establish the model shape. The methods express the functionality and are the base for building the feature list. A feature in FDD is a client-valued function. The features of the feature list are prioritized by the team. The feature list is reviewed by domain members [PCL99].

FDD proposes a weekly 30-minute meeting in which the status of the features is discussed and a report about the meeting is written [PCL99]. Reporting can in the broadest sense be compared with requirements tracking/management.

3.7 Dynamic Systems Development Method (DSDM)

DSDM was developed by a UK-based Consortium of organisations with internal IT departments or IT-based companies. It provides a framework for rapid application development. DSDM consist of five phases as shown in Figure 9. The

first two phases are done sequentially and only once. The last three phases are iterative and incremental. During these phases the development work is done. They can overlap and merge in different ways depending on the project [Sta95]. Iterations in DSDM are called "timeboxes". A timebox lasts from a few days to a few weeks. The time and the tasks are planned beforehand [PAW02].

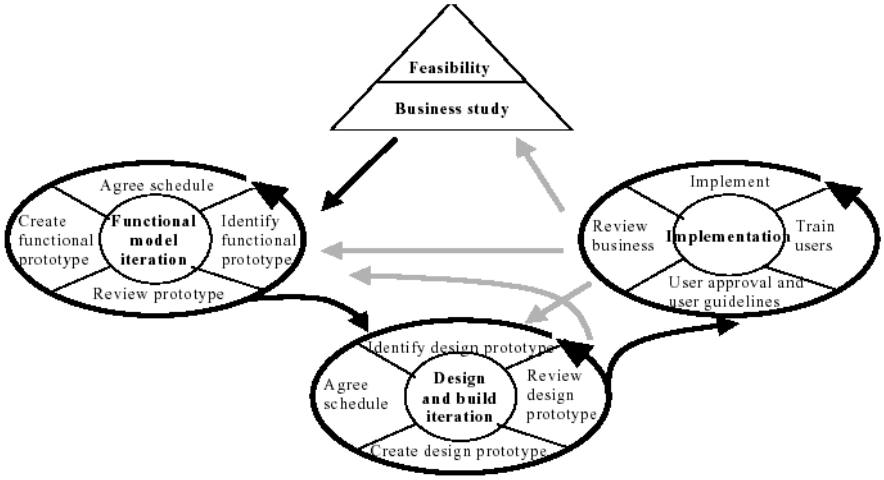

Figure 9: DSDM process [Sta95]

The Following five phases are described in detail as in "DSDM - Dynamic Systems Development Method (Jennifer Stapleton)" [Sta95]:

The feasibility study The feasibility study points out whether the project is feasible and if DSDM can be used for the planned project. It answers some basic technical questions and shows the risks of the project. The feasibility study is short and compressed. It should not last longer than a few weeks. The goal is to have sufficient information to make a decision but no more. The output is the feasibility report and an outline plan for development. Optionally a prototype can be produced to figure out if the project is technical feasible. This is only necessary if the technology is not tried or tested within the organisation.

The business study The goal of the business study is to gain understanding of the business and technical constraints. A short series of workshops is held, with the Business Area Definition as result. The Business Area Definition does not only identify the business processes but also contains information about the affected classes of users. Other outputs are the System Architecture Definition and the Outline Prototyping Plan. The system architecture definition is a first sketch of the system and can change during the project. The prototyping plan covers all prototyping activities for the following phases and a plan for configuration management.

Functional model iteration In the functional model iteration analysis documentation and prototypes are developed. The Functional Model built in this iteration consists of analysis models and software components containing the major functionality. They will also cover some non-functional requirements. Testing the software components is also an essential part of this iteration.

Design and build iteration During the design and build iteration the system is developed for operational use. The output of this phase is a Tested System not fulfilling all requirements identified but a least the requirements that have been agreed upon. The prototypes produced are reviewed by the users. The users' comments are the basis for further development.

Implementation The system is installed in the actual production environment; users are trained and the system is handed over to them. This process can be iterated if there is a large number of users. Additional to the running system the output of this phase is a User manual and a Project Review Report summarizing the outputs of the project based on the results. If there are no further requirements discovered during development and the system fulfills all requirements no further work is needed. Otherwise the process may run again from start to finish or from the functional model iteration onwards.

During the first two phases the base requirements are elicited. Further requirements are elicited during the development process. DSDM does not insist on certain techniques so any RE technique can be used during the development process. Testing is a major principle in DSDM. Testing is integrated throughout the lifecycle and not at the end of the lifecycle which reduces the possibility of major mistakes. The philosophy of DSDM is 'test as you go' [Sta95]. All sorts of testing (technical, functional) is carried out incrementally by the developer and the users on the team. DSDM explicitly highlights the use of JAD sessions and emphasizes prototyping [Sta95].

3.8 Adaptive Software Development (ASD)

ASDs principles come from research on iterative development. ASD provides a framework for developing large, complex systems with enough guidance to prevent projects from falling into chaos. The method encourages incremental, iterative development with constant prototyping. The ASD process contains three non-linear, overlapping phases: Speculate, Collaborate, and Learn (see Figure 10).

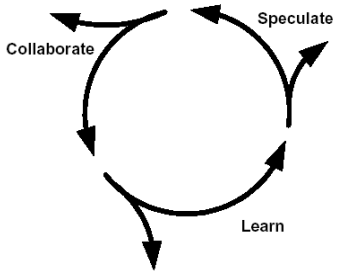

Figure 10: ASD process [Hig96]

Highsmith states that the names of the phases emphasize the difference from traditional software development. "Speculate" instead of "Planning" because planning always indicates that there are no uncertainties. The importance of teamwork is highlighted by "Collaborate". In an unpredictable environment

people need to communicate to be able to deal with the uncertainty. "Learn" points out that requirements change during development and the need to acknowledge this. In ASD deviations are not a failure of the system but will lead towards the correct solution.

Each project starts with defining the project mission, a short statement pointing out the course of the project. The mission aspects are defined in three items: a project vision charter, a project data sheet and a product specification outline. During the project initiation phase the overall schedule as well as the schedules and objectives for the development cycles are defined. A cycle in ASD lasts between four and eight weeks. As ASD is component-oriented rather then task-oriented it focuses on the results and their quality. The next phase is the adaptive cycle planning that contains the Collaborate Phase where ASD addresses the component-oriented viewpoint. Planning cycles are repeated when new requirements occur and components have to be refined. Reviews in presence of the customer demonstrate the functionality of the software during the cycle. Since these reviews are relatively rare ASD recommends JAD sessions to increase customer involvement. The final Q/A and release stage is the last phase in an ASD project. There are no proposals for how this phase is to be carried out, but emphasis is on the importance of capturing the lessons learned.

Like other agile processes, ASD is based on iterative development cycles. ASD highlights that a serial waterfall method only works in a well-understood and well-defined environment. But changes are frequent in software development and it is important to be able to react to changes and to use a change-tolerant method. The first cycles of the project should be short to ensure that the customer is involved early and to confirm the project's viability. Each cycle ends with a review. The tool for this reviews are customer focus groups. During the review meetings a working application is explored. The results of the meetings are documented change requests.

Although ASD explicitly refers to JAD sessions it does not propose schedules for holding JAD sessions.

4 Requirements engineering techniques for agile methods

4.1 Overview

Nearly all requirements engineering techniques described in chapter 2 can be found in the different agile approaches, though they may not be used in the same area or to the same extent. This chapter describes the way in which agile approaches incorporate these techniques and gives a short summary of the analysis.

4.2 Customer involvement

A key point in all Agile Development techniques is to have the customer 'accessible' or 'on-site' (XP). This is the basis for fast feedback and communication. Both leads to better understanding of requirements (on developer side) and development process (on customer side). Involving the customer is the first goal in agile approaches [1]. XP, for example, assumes an "ideal" user representative: the representative can answer all questions correctly that the developers may have, s/he is empowered to make binding decisions and able to make the right decisions.

Stakeholder involvement is also a key point in current Requirements Engineering work. Stakeholders own the knowledge about the system to be developed. The different RE elicitation techniques aim to get as much information as possible about the system from all people involved (users, domain experts, etc.). In 1995, the CHAOS report from the Standish Group [Gro95] showed the critical importance of customer involvement. Customer involvement was found to be the number one reason for project success, while the lack of user involvement was the main reason given for projects that ran into difficulties. Thus, traditional RE and agile methods agree on the importance of stakeholder involvement. The difference between traditional approaches and agile methods

[1] It should be mentioned that in this context a 'customer' is comparable to a 'user'. Traditionally, in RE the customer is the one paying for a software system and the user the one using this software system.

is that in traditional approaches the customer is mainly involved during the RE phase and not during the actual software development while agile methods involve the customer throughout the whole development process.

Although agile methods are based on strong customer involvement during the whole development process, there is still a problem. In most cases, the methods ask for several customers with different backgrounds, but sometimes only one customer is available to work with the development team. This means that not all the questions arising can be answered in enough detail. This may result in delays during development or in developers making potentially incorrect decisions. Furthermore, some business areas may not be sufficiently understood by the customer, so s/he can not tell the development team about the related requirements. Even if the requirements are elicited in group sessions (DSDM, Scrum) it is not guaranteed that users or customers with the necessary background will be present. But the same problem arises in traditional requirements elicitation. Although there are many different techniques to elicit requirements no one can ensure that all problem areas are covered. Other techniques like reviews or tests are necessary to ensure that these problem areas are discovered.

4.3 Interviews

As customer involvement is the primary goal of agile software development, the most common RE-related technique are interviews. Interviews provide direct and 'unfiltered' access to the needed knowledge. It is generally known that chains of knowledge transfer lead to mistakes and misinformation. Developers who have direct in contact with the customer derive information with less errors. Personal communications establish relationships between customers and developers that increases confidence and trust. All agile approaches emphasize that talking to the customer is the best way to get the information needed for development and to avoid misunderstanding. If anything is not clear or vaguely defined, customers or developers should try to communicate with the person responsible to avoid indirect transfer of knowledge specifically.

4.4 Prioritization

Prioritization can be found in all the agile approaches. A common practice is to implement the features with the highest priority first to be able to deliver customers the most valuable software. During development the knowledge and understanding of the project increases and new requirements are added. Therefore, prioritization is repeated frequently during the whole development process to keep priorities up-to-date. Prioritization helps customers to understand the differences between requirements and in which way they could affect the system.

4.5 JAD

JAD sessions are used in ASD to increase customer involvement. During the JAD sessions developers and customers discuss the desired product features. This kind of discussion can be very productive as the session leader prevents the participants from 'running off course'. In addition, focusing the interaction to short JAD sessions does not bind the customer representative as much as the on-site customer in XP. The results of the sessions are usually documented and accessible if further questions arise later. The documentation requirement could be relaxed in an agile context. To have constant feedback, JAD sessions should be held frequently during the development process.

In DSDM, JAD sessions are used to gain understanding of the new system at the beginning of a project. JAD promotes cooperation, understanding and teamwork among the different groups of participants. As participants should come from different backgrounds, information considering various parts of the new system can be gathered and provide a base for further requirements elicitation. They also encourage customer involvement and trust. This is clearly in line with agile principles.

4.6 Modeling

Although modeling is used in AM, the purpose is different. In AM, models are for example used to communicate understanding of a small part of the system

under development. These models are mostly throw-away models. They are drawn on a whiteboard or paper and erased after fulfilling their purpose. Most of the models do not become part of the permanent documentation or final model of the system. When modeling is used in RE, there are different models starting with an overview of the whole system followed by other models which go into more detail. All these models normally become part of the system documentation and need to be kept up to date.

The purpose of modeling in FDD is the same as in RE: The developed model represents the whole system and development is based on this model. But as the design and build phase are iterative, changes to requirements, and in consequence to the models, can be made later in development process. The model represents a first step to start development but does not completely describe the final system.

4.7 Documentation

Complete and consistent Requirements Documentation is barely represented in agile software development. Some agile methods include a kind of documentation or recommend the use of a requirements document (DSDM, Scrum, Crystal) but the decision of the main contents or extend is left to the development team and not described in detail.

The lack of documentation might cause long-term problems in Agile Development but provides speed ups in the original development. Documentation is used for sharing knowledge between people. A new team member will have many questions regarding the project; These can be answered by reading and understanding appropriate documentation. Asking other team members may slow down work, because of the time required to explain a complex project. The same situation arises when changes are to be made to a finished project. It is possible that the developers originally working on the project are not available (moved to another company or are working on a new project) and a new team has to make the changes. Long term problems are reduced as agile methods often produce clean and compact code. Additionally, customers often ask the

agile team to produce design documentation before the team is resolved when the system moves to a pure production environment. The scope of the documentation is often very limited and focuses on the core aspects of the system. This increases the chances that the documentation can be kept up to date when the software is changed.

On the other hand, agile methods should be more productive in terms of code than traditional approaches as they spend less time one producing documentation than traditional RE approaches. Agile methods tend to err on the side of inadequate documentation while traditional approaches tend to overdocument.

Traditional approaches try to produce enough documentation to be able to answer the majority of all questions that might be asked in the future. To be able to do so, they need to (1) anticipate future questions and (2) answer them in a concise and understandable manner. Both aspects are difficult - this is why writing good requirements documents is so hard. Usually, to get a good coverage of future questions, traditional approaches might produce too much documentation (i.e. answering questions that will in fact never be asked and wasting money on writing the answers). This results in problems with keeping all documents up-to-date and in problems of readers in locating the relevant information for their current questions.

The advantages or disadvantages of documentation are related to team size. With a big team, it might be better to have documentation instead of explaining the same thing multiple times to different people. When agile methods try to scale up to larger teams, they often increase the amount of documentation to be produced compared to small projects [Coc02].

4.8 Validation

Requirements validation plays an important role in all agile software development approaches. The software is validated by review meetings and acceptance tests. This increases the customers' confidence in the program and the development team because the system is demonstrated to work in envisioned capacity. Review meetings also show that the project is in scope and schedule.

All approaches have a similar way to validate requirements. They use different kinds of review meetings to present the developed software. The customers can use the software, experience how the system works in detail and which functionality has already been implemented. If they have questions, these can be immediately answered by the developers which are present at each review meeting. Customers can also discuss the way features have been implemented with developers and ask for changes in design. During the review meetings they learn about the strength, weaknesses, advantages and limits of the design and technology. The customers can see if the requirements implemented were understood, and consequently if the were implemented correctly. Customers can also run acceptance test to validate if the system reacts in the expected way and, if not, clarify the problem. The questions and change requests influence further development. Depending on the method and project circumstances, the software is put into production after each review. This fast deployment changes the economics of the software project by providing a faster return on investment.

Especially in XP, where the customer should always be present during development, the possibility of being able to ask questions and to see that the tests are passing increases trust in the development team and the software.

As all agile approaches are based on small releases, validation is strongly enforced. At least with every new release, developers get feedback from the customer. This kind of development is comparable to evolutionary prototyping which are both based on the same idea of delivering frequently a piece of working code to enhance feedback. The difference lies in the strong emphasis on testing in agile approaches.

4.9 Management

Limited Requirements Management is a common 'attribute' of the agile software development methods. From the RE view it should be possible to track changes made to requirements, design, or documentation to see why any changes were made. But tracking changes results in more work and additional documentation. Additionally, it has not yet been shown that tracking changes provides any

monetary benefit for a project.

Lack of documentation could be critical if the requirements form a legal contract between two organizations. That is the reason why agile project contracts often are based on time and expenses and not on fixed price/fixed scope. Contracts based on fixed price/fixed scope create trust by defining what the customer will get, for what price and in which time. Agile approaches try to create trust by working closer with the customer. As the customer sees that the development team is working on the software and not doing something else, s/he believes that the team will deliver working software that will meet his/her needs.

Nevertheless, agile methods provide a good base for Requirements Management. All the requirements are written on index cards (the user stories) or maintained in a product backlog. The difference is the level of detail in which a requirement/user story/feature is described: agile practices usually omit the details and let them be clarified when the requirement is actually tackled during software development: they postpone the expenses for gathering the details until the requirement needs to be fulfilled in the next iteration. One can call this lazy requirements elicitation.

DSDM provides the possibility to track changes. When parts of the business study have to be repeated, these changes can be documented in a separate document. The Product Backlog in Scrum could be used to version changes made to requirements. Instead of deleting old versions they could be kept referring to the new requirement or with a note why they were deleted/changed.

4.10 Observation and Social Analysis, Brainstorming

These techniques are not explicitly mentioned in any agile software development method. But as they are the fundamental techniques in the requirements elicitation process, they can be used with any of the agile approaches. Observation methods can espacially provide benefit in requirements elicitation as a high qualified 'user' is not always the best person to talk about her/his work process. These people sometimes forget important details as the work has become

routine. These details can be discovered during observation.

4.11 Non-functional requirements

In agile approaches handling of non-functional requirements is ill defined. Customers or users talking about what they want the system to do normally do not think about resources, maintainability, portability, safety or performance. Some requirements concerning user interface or safety can be elicited during the development process and still be integrated. Non-functional requirements often tend to be related to functional requirements and are difficult to express. But most non-functional requirements should be known in development because they can affect the choice of database, programming language or operating system. Agile methods should include more explicitly handling non-functional requirements. But also requirements engineering does not define an explicitly handling of determining non-functional requirements.

4.12 Conclusion

The RE process phases Elicitation, Analysis, and Validation are present in all agile processes. The techniques used vary in the different approaches. As the phases are not as clearly separated as in the RE process, they also merge in some ways (the Planning Game in XP is an elicitation and analysis technique). They are also repeated in every iteration which makes it harder to distinguish between the phases and therefore to compare them to the RE process phases. The techniques used in Agile Development processes are sometimes described vary vaguely and the actual implementation is left to the developers. This is a result of the emphasis on highly skilled people in agile methods: "good" developers will do the "right thing". Traditional approaches, on the other hand, prescribe details of what needs to be done and so provide every developer with more guidance to do the "right" thing. Unfortunately, determining upfront what the right thing in a given project context will be is very difficult.

As RE Management is based on documents, it is not very well represented in agile software development approaches. Documentation is a part of agile

software development but not to the same extend as in RE. As all agile approaches include at least a minimum of documentation it is the responsibility of the development team to write enough documentation for further development.

Overall, in key areas (like stakeholder involvement) agile methods and RE are pursuing similar goals. The major difference is the emphasis on the amount of documentation needed in an effective project. This partially steams from differences in core assumptions on the stability of requirements. Thus, the two approaches are more like Whisky & Ice than Fire & Ice.

5 Objectives of development

5.1 Overview

The literature survey showed in which way RE techniques are used within agile agile software development. Another possibility would be to use some of the RE techniques within a defined process like a management tool for software development. Several of the RE techniques could be integrated in such a process. Different kind of models, for example, are already developed with the help of computer programs and could be included easily. But the different models are not always used in the same way and in all agile approaches. Prioritization, for example, is a base technique in all agile approaches and always used in the same way as it is to prioritize the tasks or requirements. As prioritization is regarded to have a substantial influence on the development process we decided to provide teams in an agile software project with the possibility to prioritize tasks within a defined process. As today teams can be distributed, the prioritization should also be possible in a distributed environment. This means that the team will not have to meet to discuss prioritization which will save time. A prioritization feature should allow each team-member to prioritize the tasks/requirements for themselves to build a basis for the 'team'-priority. This team priority could first be calculated based on the team-member priorities. Using this calculated team-priority as a starting point, the whole team can reach consensus on a final priority.

An existing management tool for agile software development projects is the MILOS ASE system developed at the University of Calgary. As it is accessible over the internet, it is especially useful in a distributed development process.

5.2 Development environment

The MILOS ASE (short M-ASE)[2] system was developed to support agile software engineering practices in a distributed environment. It is based on the MI-

[2]MILOS Agile Software Engineering

LOS[3] project of the University of Calgary and the University of Kaiserslautern. The aim of MILOS is to support planning, managing, and executing software projects and to provide mechanisms to access knowledge and experience during planning and execution. The main difference between MILOS and M-ASE is that MILOS was developed for traditional software engineering and therefore focuses on in- and outputs to processes while M-ASE focuses on iterations and tasks and their planning and management. An online demo of M-ASE is accessible under 'http://sern.ucalgary.ca/~milos'. The main features of M-ASE are iteration planning, user story management, progress tracking, and metrics collection.

An iteration is a fixed period of time during which developers will be working on the project. Before developers start working, iterations have to be created and planned. When an iteration is created it will be given a default name generated with the given dates or an alternative name. A general description of the iteration, for example goals or task types, an estimation of effort in work-hours needed to complete the iteration and a list of technologies which will be used during the iteration can be added. The effort is determined based on work-hours and the velocity of the last iteration. The iteration effort is calculated based on task efforts. For each member on the team the time they plan to spend on the project (in general the number of hours the developer is paid to work per day, times the number of days the iteration lasts) can also be added. Each iteration consists of user stories and other tasks which are assigned to the iteration.

User stories (or story cards) are a description of a functionality which is to be implemented in the software product. Each user story should have a short name describing the functionality and an estimated 'initial effort' in working hours. To be able to track the user story it should be assigned to a project as well to an iteration. More details like the type of activity or risk can be added to the user story to improve iteration planning and performance. The user stories contain a special description field being the most important field of

[3]Minimally Invasive Long-term Organizational Support

the user story. This field should be used to add all information available about the functionality to the user story.

The 'Planning Whiteboard' in M-ASE is used for the management of user stories. Management includes assigning user stories to team members, providing estimates for user stories and creating subtasks to an user story before the beginning of the iteration. When a new user story is created it is automatically assigned to the project manager. As there are normally different people working on the project and everyone will be responsible for several tasks the default assignment should be changed on the whiteboard before beginning the iteration. The whiteboard allows to add subtasks to an user story which breaks the story down into steps. This can make it easier for developers to estimate and understand the user story. All initial estimates entered when the story card was created can be updated within the whiteboard before the iteration begins. Setting these initial estimates finally indicates the begin of the iteration. When an iteration is completed and there are still unfinished tasks, these tasks can be copied to the next iteration.

5.3 Objectives

As described in chapter 4.4, prioritization is an important technique both in Requirements Engineering and in Agile Software Development. Prioritization is still poorly represented in M-ASE. It is possible to say if a task is very important or not important but there is no possibility to rank the tasks within an iteration. Prioritization in a distributed team would be an important feature in M-ASE.

The basic principle in agile development is to deliver valuable software to the customer frequently. But in most cases the developer can not decide what is of most value for the customer as the customer is the one owning the knowledge about the business. The prioritization feature in M-ASE will enable users to rank tasks within an iteration. As customers and developers have access to the system, there will be business influences and technical influences on the prioritization. This will lead to a ranking that does not only provide the most value for customers, but is also based on technical realisation . The resulting

team-priority will help developers to implement a system giving customers the most value and including those features which are implementable in a reasonable period of time. As the project goes ahead, the features with 'less' value or which are harder to implement will be prioritized higher, as all other features have already been implemented.

The prioritization feature will allow the customer to cancel the project at any time, but still get a valuable and functioning software system. The final prioritization also will all users to discuss the tasks and their priority. This will help to increase understanding of the problem area.

6 Design and implementation

6.1 Overview

This chapter describes how the prioritization feature was integrated into the M-ASE system. First the user-interface and functionality is explained from a user view. The implementation description goes into more detail and explains the technical realization.

6.2 Functionality

The prioritization feature is accessible over the site displaying the iteration details (see figure 11). The site contains a link to the prioritization site and an overview table. The overview table shows all tasks in the chosen iteration and their team-priority. The table also contains an information field. In this information field a short message is displayed telling the user, if all team-members have prioritized the tasks in this iteration.

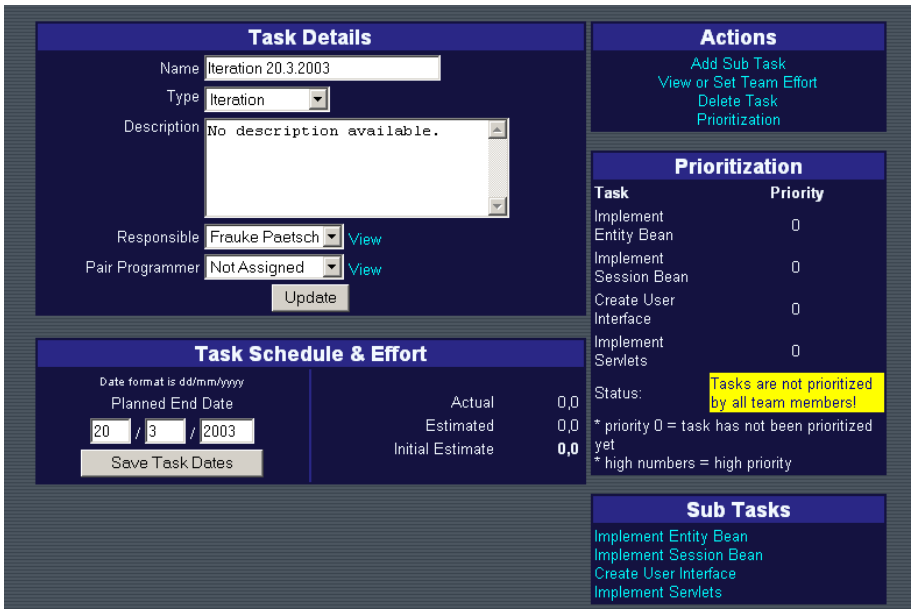

Figure 11: Iteration details site

The prioritization site is only accessible when tasks have been entered. It displays a table with the tasks and team-priority and a table containing all team-members (see figure 12). The priority displayed is the team-priority which has been calculated based on the team-member priorities. After the team has agreed on a final team-priority, this final priority will be shown in the table. If the tasks have not been prioritized or new tasks have been added to the iteration, the tasks will have the priority '0'. The table also contains an information field, in which messages can be displayed. These messages will inform the user after prioritizing, if, for example, two tasks have the same priority or, if the resulting team priority contains equal values. If a team member changes priorities or if tasks are added after the team has agreed on the final priority, the project-manager will be displayed a message in the information field. The project-manager will be asked if the final prioritization should be repeated.

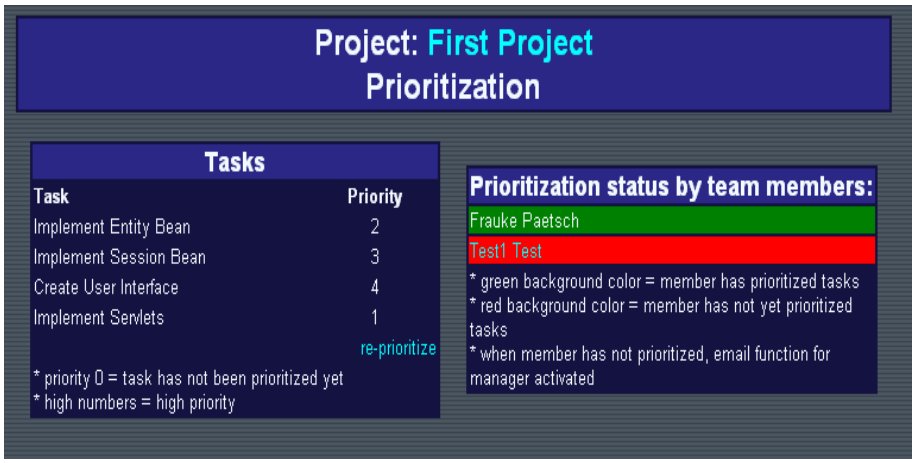

Figure 12: Prioritization main page

The second table on the site displays all team-members. The background of the table-cell is either green, indicating that this team-member has prioritized the tasks in the iteration, or red, indicating that the team-member has not prioritized the tasks in the iteration. If the current user is the team-manager, the names of those team-members not having prioritized will be a link to a

'reminder-email'.

The first step in prioritization is that team members prioritize the tasks per iteration on their own (see figure 13). The prioritization can be repeated at any time as changes may be necessary if new tasks are added, tasks are deleted or new facts increase or decrease a task's priority. These changes effect the team-priority but not the final team-priority when the final team-priority has already been finalized. Changes to priorities, including the initial prioritization, are made using the prioritization main page. If the team member wishes to change priorities this can be done by following the 're-prioritize'-link which will be displayed within the table.

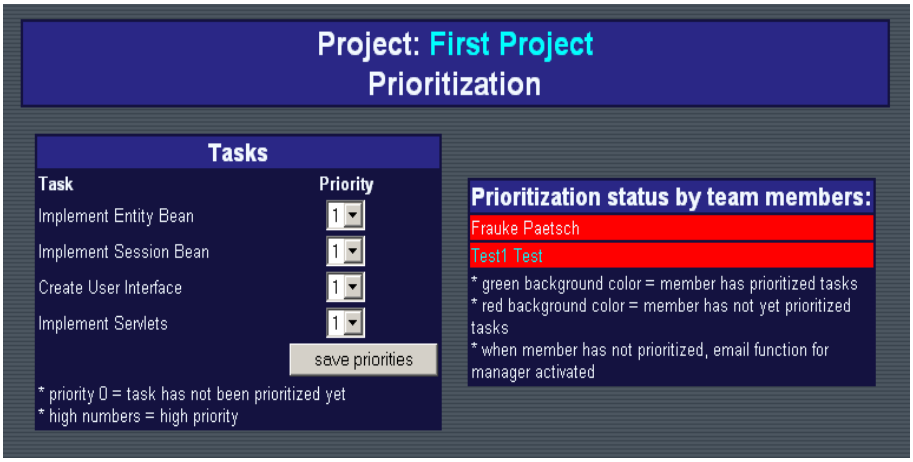

Figure 13: Prioritization be team member

The site to (re-)prioritize displays the same two tables as the previous site, but the priorities are now shown in drop-down-lists. If the tasks have not been prioritized, all tasks will have the priority '1'. If the tasks have been prioritized, the priority displayed is the team-priority. The numbers in the drop-down-lists start with 1 and go up the number of tasks in the iteration. The higher the number, the higher the priority. The user selects a priority per tasks and saves the data by clicking on the 'prioritize'-button.

The final ranking should be done during a team-session (for example during

a NetMeeting-Session) where all team members can discuss the ranking. The ranking is done on the planning whiteboard (see figure 14). The ranking can

Figure 14: Planning Whiteboard

only be done if the tasks are shown in order of priorities. The tasks are shown in the order of the team-priority. Every task can be moved one step upwards/-downwards or to the top/bottom of the list using arrows displayed next to the task-priority. When the priority of a task has been changed, the site is reloaded and shows the new order and priorities. If the tasks are displayed ordered alphabetically, only the priority, but not the arrows to move the tasks, will be shown. When the team has agreed on a final ranking the project-managers saves the data by checking the check-box next to the iteration, of which the priorities are to be finalized, and clicking on the 'finalize priorities'-button. Only the project-manager can finalize the team-priority.

6.3 Implementation

The first step was to implement a session bean and an entity bean to provide access to the database, the algorithm to calculate the team-priority and other functions used on the whiteboard or the prioritization site (see the class diagram in figure 15). The entity bean provides the access to the database while the session bean provides the business logic. The functionalities provided by the session bean include:

- adding new rows to the database-table after checking that the data is not already in the database
- updating existing rows
- checking if two tasks have the same priority
- calculating team-priority
- updating team-priority
- changing priorities (when task are moved up/down on the whiteboard)
- saving final team-priority

The team-priority is calculated based on the individual team-member-priorities. For every task, the priorities of all team-members are summarized. The resulting team-priority is the new order of the summarized priorities, with the lowest value indicating the lowest priority (see listing 2). The team-priority is updated every time a team-member re-prioritizes the tasks. As long as the team-priority has not been finalized, also the final team-priority is updated. The overview tables and the whiteboard always display the final team-priority.

When tasks are move upwards/downwards on the whiteboard, the final team-priority is changed. That means that all logged-in team-members will be able to see the changes after reloading the website. The function for moving the tasks uses the Java LinkedList to change the order (see listing 1). All the tasks are stored ordered by priority in the LinkedList. The index of the task to be moved

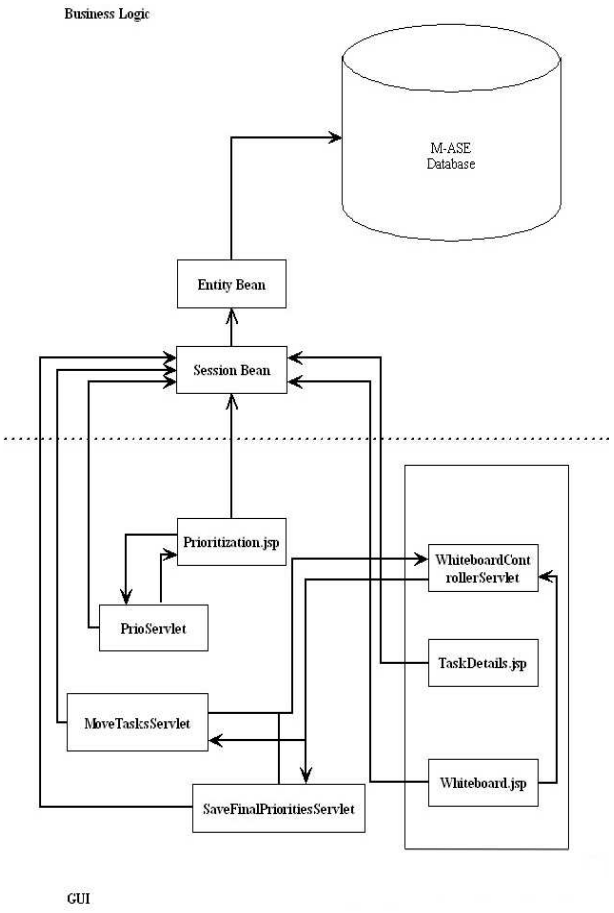

Figure 15: Class Diagram

is saved in a variable while the tasks are stored in the LinkedList. When all the tasks have been stored, the task to be moved is removed of the list and included at the new position. Then all the tasks are re-prioritized according to their new order in the LinkedList.

When the final team-priority is saved, a boolean field in the database (finalized) is set to 'true' indicating that the final team-priority will not be changed any more. If a team-member re-prioritizes the tasks another boolean field int the database (changed) will be set to 'true' and the project-manager will have to decide whether the final prioritization sit to be repeated or not. If s/he decides

to re-prioritized the boolean field 'finalized' and the boolean field 'changed' will be set to 'false' and the final prioritization can be repeated. If the project-manager decides not to re-prioritize, only the boolean field 'changed' will be set to 'false'. These boolean fields are checked every time one of the jsp-sites is loaded and the appearance of the site is based on these checks.

One additional jsp-site was added and two jsp-sites had to be changed. The new jsp-site is the prioritization-site. The site contains the two tables, one displaying the tasks with priority, the other one displaying the team-members. The tasks and team-members are displayed in the order they are stored in the database. The message displayed in the tasks table and the appearance of the site depends on boolean fields in the database and URL-parameters. If tasks have not been prioritized, the boolean field 'prioritized' is set to 'false'. This field is checked, when the site is loaded and depending on the status (true/false) whether the overview table or the table with the drop-down lists is displayed. The table containing the drop-down lists is also displayed when the URL-parameter 're-prioritize' is set to 'true'. If a team-member has chosen a priority twice or the calculated team-priority contains the same priority twice, a URL-parameter is set and a corresponding message is displayed. URL-parameters will be set by the PrioServlet. When the user saves the priorities, the data is sent to the PrioServlet to check for duplicate priorities and to calculate the team-priority. Depending on the return-value of the functions, the PrioServlet redirects to different URLs.

On the jsp-site showing the details of an iteration, a table displaying all tasks of the iteration with their priority was added. As the jsp-site is also used to show task details, it has to be checked whether the current process is a task or an iteration. When the data for the table is read out of the database, it is simultaneously checked if all team-members and tasks are in the prioritization table (see listing 3).

On the whiteboard jsp-site, functionality to move the tasks up/down when the team agrees on the final team-priority was added. The tasks are moved using java-script functions which forward the iteration id, task id and a parameter

with the function-name to the WhiteboardControllerServlet. Depending on the function-name, the controller servlet forwards the data to different, more specialist, servlets. Two servlets were added to cope with the functionality used to move the tasks and to save the final team-priority. Both servlets use functions in the session bean and redirect back to the whiteboard jsp-site (see listing 4). They do not include business logic and are only used to forward the data.

Additional to the user-interface and the Java beans, an new test-class for the session bean was included in the M-ASE system. The test-class checks the basic functions of the session bean, e.g. adding a task to the database and removing a task from the database (see listing 5). If the test-class can not add or remove a task, an exception is thrown and the developer running the test can check what went wrong.

7 Summary

A survey of the most common requirements engineering techniques and their description built the basis for the research on different agile software development approaches. The agile approaches were studied in regard which of these RE techniques are already used and which could additionally be used. The main difference between requirements engineering and the agile software development approaches is the amount of documentation and the basic ideas behind both techniques. In requirements engineering requirements are regarded as stable and are therefore documentable before software development. In agile software development requirements change during the whole development process and as this would result in permanent changing of the requirements document documentation is reduced as much as possible.

Both approaches are based on customer involvement and include prioritization of requirements. Prioritization is used to help developers implement the features most valuable for the customer first. As this technique provides a lot of benefit, a feature to prioritize tasks was developed for an online management system called M-ASE for agile software development. The feature allows team members to prioritize tasks per iteration. A team priority is based an the average of the individual priorities. The team can finally agree on a ranking for the whole team.

As the team priorities are based on the average of the individual priorities, future work on M-ASE could include alternation of the calculation and the use a mathematical algorithm like AHP. This would result in more user input, but could make it easier to decide on a ranking. A possible extension to the prioritization feature could be a 'discussion board' where the team members can comment the tasks and give reasons for the chosen priority. This can be useful if the team members are not able to communicate via NetMeeting or similar tools.

A Enterprise Java Beans (EJB 2.0)

EJBs are a specification for creating deployable server-side components in Java. The specification describes the component architecture with agreed interfaces which enables components to be run in any application server supporting EJB. The component architecture is a consistent framework for creating distributed n-tier middleware. But as the application server handles everything concerning network and distributed components developers can concentrate on the business logic and do not have to worry about how to code a complex distributed component framework or distributed components. EJB components are deployable and can be imported and loaded into an application server where these components are hosted. EJB is designed to be able to write scalable, reliable, portable, reusable, and secure applications which can be used across any vendor's enterprise middleware services.

A typical component architecture consists of (see figure 16):

- EJB Server
- EJB Container
- Home Interface
- Remote Interface
- Enterprise Java Bean
- EJB Client

The EJB Server provides system services like an execution environment to run the EJB Container within or naming and transaction services. The server also provides an organized frame work, multiprocessing, load-balancing, and device access. EJB container running within the server are visible to the outside world.

EJB Container act as the interface between an Enterprise Java Bean and the outside world. Any access to an bean is handled through the container-

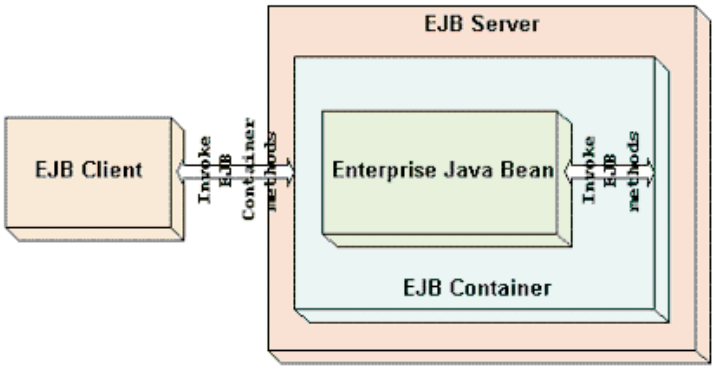

Figure 16: EJB component Architecture

generated methods. A client accesses only these methods which invoke the bean's methods.

The Home Interface defines methods for locating, creating or removing instances of EJB classes. The home object is the implementation of this interface. The implementation is generated automatically by the EJB Container. The Remote Interface is a list of methods available for the Enterprise Bean. The EJB Object implements the remote interface and is also generated by the container.

The Enterprise Java Bean should never be accessed directly by anyone but the EJB container. This is guaranteed by the use of the remote interface for which the implementation is generated. There are three types of EJBs:

- Session beans and
- Entity beans
- Message-driven beans

Session beans are usually associated with a specific EJB Client. The session bean is created and destroyed by this client. When the system crashes, the session bean is destroyed. Session beans normally model business processes and can either have states or can be stateless.

As entity beans represent business data they always have a state and can be shared by multiple EJB Clients. The state can be stored which allows entity beans to survive System Shutdowns.

The difference between message-driven beans and session beans is, that message-driven beans can be called by sending a message to the bean.

EJB Clients locate the EJB Container containing the Enterprise Java Bean they want to access through the Java Naming and Directory Interface (JNDI). This container is then used to invoke the beans methods. The client only gets a reference to an EJB Object and not a reference to the actual Enterprise Bean. To locate, create, or destroy instances of an Enterprise Bean the home object is used. The EJB Object is only used to invoke the business methods of a bean instance.

Using EJBs has several advantages. The three most important ones are:

EJB is agreed industrial specification This will help for example to find business partners since the technology will be compatible.

EJB simplifies portability The specification is published and available freely. As it is a standard it's architecture does not depend on the vendor. (WORA - Write Once, Run Anywhere)

EJB speeds up development Middleware does not have to be constructed as it is provided by the application server.

The programming language used for the component architecture is Java. This has some reasons. First, developers want to separate the implementation from declaration (interfaces). Customers need to know what a component can do, but they should not and do not need to know how the component works. Java supports this kind of separation via the class and interface keywords. Second, Java was designed to be safe and secure. There are for example no pointers. Also Java has a rich set of libraries which avoids to reinvent structures in a worse way. Third, as Java is a platform independent programming language this enables EJBs to be run cross-platform.

EJB uses lots of different Java technologies which are all part of J2EE. J2EE is a robust suite of middleware services. The services provided and used with EJB are:

Java Remote Method Invocation (RMI) Specifies communication between distributed objects

Java Transaction API (JTA) Java Transaction Service (JTS) Allows components to be bolstered with reliable transaction support

Java Messaging Service (JMS) Communication via messaging

Java Servlets Networked components used to extend the functionality of a Web server

Java Pages (JSP) Similar to servlets. The difference is that JSPs are not pure Java code and more centered around the look-and-feel issues.

JavaMail Service to send email

Java Naming and Directory Interface (JNDI) Access to naming and directory system

Java Database Connectivity (JDBC) API for accessing relational databases

In general, EJBs are used to help solve different kind of business problems. This might be some special business rules or business logic, or they are used to access a database or another (distributed) system. EJBs are NOT GUI-components, rather they 'sit' behind the GUI and do their work without interfering with the GUI. The GUI can be a thick client, a dynamically generated web page, or XML-based Web Service wrappers.

EJBs can be regarded as a system with different roles. All the roles have a special functionality or task. These roles are [ERJ02]:

The Bean Provider , who supplies business components or enterprise beans

The Application Assembler , who writes an application to put all the different components together

The EJB Deployer , who deploys the written application in a running operational environment

The System Administrator , who is responsible to upkeep and monitor the deployed system

The Container And Server Provider , who supplies the application server

The Tool Vendors , who build tools to easily manage the whole EJB development and deployment.

B Listings

```
public void moveTask(Long taskID, Collection subProcessC,
 String direction)
 {
    LinkedList ll = new LinkedList();
    TreeMap tm = new TreeMap();
    Iterator it;
    int index = 0;
    int saveIndex = 0;
    ProcessSession pSession = null;

    it = subProcessC.iterator();

    try
    {
        pSession = EJBRemoteHomeCache.getInstance().
        getProcessSessionHome().create();

        //order tasks by priority
        while(it.hasNext())
        {
            ProcessVO tempProcess = (ProcessVO)it.next();
            long priority = tempProcess.priority;
            Long id = tempProcess.id;

            tm.put(new Long(priority), id);
        }

        //put tasks in linkedlist
        Set set = tm.keySet();
        it = set.iterator();
        while(it.hasNext())
        {
            Long id = (Long)tm.get(it.next());
            if(id.equals(taskID))
                saveIndex = index;
            ll.add(id);
            index++;
        }

        if(direction.equals("down"))
            ll.add(saveIndex+1, ll.remove(saveIndex));
        else if(direction.equals("downdon"))
            ll.add(index-1, ll.remove(saveIndex));
        else if(direction.equals("up"))
            ll.add(saveIndex-1, ll.remove(saveIndex));
        else if(direction.equals("upup"))
            ll.add(0, ll.remove(saveIndex));
```

```
        it = ll.iterator();
        index = 1;
        while(it.hasNext())
        {
            ProcessVO tmpProcess = pSession.
            getProcessByPrimaryKey((Long)it.next());
            tmpProcess.priority = index++;
            pSession.updateProcess(tmpProcess);

        }
    }
    catch (CreateException e)
    {
        e.printStackTrace();
    }
    catch (FinderException e)
    {
        e.printStackTrace();
    }
    catch (RemoteException e)
    {
        e.printStackTrace();
    }
}
```

Listing 1: Move Tasks

```
//Method calculateTeamPriority
//@param iterationID
//@throws RemoteException, FinderException
public boolean calculateTeamPriority(Long iterationID)
throws FinderException, RemoteException
{
    Collection c = null;
    Iterator i = null;
    Long agentID = null;
    boolean totalPrioErr = false;

    try
    {
        ProjectVO ProjectVO = null;
        ProjectSession pjSession = EJBRemoteHomeCache.
        getInstance().getProjectSessionHome().create();
        ProcessSession pSession = EJBRemoteHomeCache.
        getInstance().getProcessSessionHome().create();
        // get members of the team
        ProjectVO p = pSession.getProject(iterationID);
        Vector vec =  pjSession.getMembers(p.id);
        Enumeration e = vec.elements();
```

```
        Hashtable hm = new Hashtable();
        TreeMap tm = new TreeMap();

        // go through the team members
        while(e.hasMoreElements())
        {
            c = null;
            i = null;
            AgentVO tempAgent =  (AgentVO) e.nextElement();

            // get member info
            agentID = new Long(tempAgent.id.longValue());

            c = findByAgentIDAndIterationID(agentID,
            iterationID);
            i = c.iterator();
            // Iterator with all tasks of this teamMember

            while(i.hasNext())
            {
                Prioritization ptemp =
                (Prioritization)i.next();

                Long taskID = ptemp.getTaskID();
                Long taskPriority = ptemp.getPriority();

                if(!hm.containsKey(taskID))
                    hm.put(taskID, taskPriority);
                else
                {
                    long l = ((Long)hm.get(taskID)).
                    longValue();
                    l += taskPriority.longValue();
                    hm.put(taskID, new Long(l));
                }
            }
        }   // end while(it.hasNext())


        Enumeration en = hm.keys();
        Vector v = new Vector();
        totalPrioErr = false;
        while(en.hasMoreElements())
        {
            Long key = (Long)en.nextElement();
            Long o = (Long)hm.get(key);

            for(int k=0; k<v.size(); k++)
            {
                if(o.equals(v.elementAt(k)))
```

```
                    totalPrioErr = true;
            }

            v.addElement(o);


            tm.put(o,key);

        }

        Set set = tm.keySet();
        Iterator iter = set.iterator();
        int x = 1;

        while(iter.hasNext())
        {
            Long taskID = (Long)tm.get(iter.next());

            c = findByAgentIDAndTaskID(new Long(0), taskID);
            i = c.iterator();
            Prioritization ptemp = (Prioritization)i.next();
            if(!ptemp.getFinalized())
                updateFinalPriorities(x, taskID);
            ptemp.setPrioritized(true);
            ptemp.setChanged(true);
            ptemp.setPriority(new Long(x++));

        }
    }
    catch (CreateException e)
    {
        e.printStackTrace();
    }
    catch (RemoteException e)
    {
        e.printStackTrace();
    }
    catch (FinderException e)
    {
        e.printStackTrace();
    }

    return totalPrioErr;
}
```

Listing 2: team priority calculation

```
//check if all tasks of iteration are in prioritization table
boolean prioFlag = false;
boolean newAgent = false;
```

```
while (subProcessI.hasNext())
{
ProcessVO ptemp = (ProcessVO) javax.rmi.
PortableRemoteObject.narrow(subProcessI.next(),
ProcessVO.class);

Collection col1 = null;
Collection col2 = null;

col1 = prioSession.findByTaskID(ptemp.id);
col2 = prioSession.findWithoutPrio(new Long(pid), ptemp.id);

if(col1.size() == 0)
{
    prioSession.addEmptyRows(new Long(ppid), new Long(pid),
    ptemp.id);
    prioFlag = true;
}

if(col2.size()>0)
    prioFlag = true;


%>

<tr>
    <td align="left"><%= ptemp.name%></td>
    <td align="center"><%=ptemp.priority%></td>
</tr>
<%
}
```

Listing 3: check prioritization table

```
public class WhiteboardSaveFinalPrioritiesServlet extends
HttpServlet
{
    protected void doGet(HttpServletRequest arg0,
    HttpServletResponse arg1)
        throws ServletException, IOException {
        doPost(arg0, arg1);
    }

    protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {


    try {
```

```
        PrioritizationSession prioSession = EJBRemoteHomeCache.
        getInstance().getPrioritizationSessionHome().create();

        Enumeration names = request.getParameterNames();
        while (names.hasMoreElements()) {
            String nextName = (String)names.nextElement();
            String nextParam = request.getParameter(nextName);

            if (nextName.startsWith("checkbox")) {
                String nextRow = nextName.substring(8);
                int nextProcId = Integer.parseInt(nextRow);

                prioSession.saveFinalPriorities
                (new Long(nextProcId));

            }
        }
    }
    catch (CreateException e) {
        e.printStackTrace();
    }

        request.getRequestDispatcher("milosMain.jsp?side=/
________MilosXP/sideLinks.jsp&content=/MilosXP/
________whiteboard.jsp").forward(request, response);

    }

}

public class WhiteboardMoveTaskServlet extends HttpServlet {

    protected void doGet(HttpServletRequest arg0,
    HttpServletResponse arg1)
        throws ServletException, IOException {
        doPost(arg0, arg1);
    }

    protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException
    {

        String taskID = null;
        String direction = null;
        String iterationID = null;
        PrioritizationSession prioSession = null;
        Collection c = null;
        boolean prioritized = false;
```

```
        try
        {
            prioSession = EJBRemoteHomeCache.getInstance().
            getPrioritizationSessionHome().create();

            Enumeration names = request.getParameterNames();
            while (names.hasMoreElements())
            {
                String nextName = (String)names.nextElement();
                if(nextName.equals("direction"))
                    direction = request.getParameter(nextName);
                else if(nextName.equals("taskID"))
                    taskID = request.getParameter(nextName);
                else if(nextName.equals("iterationID"))
                    iterationID = request.getParameter(
                    nextName);

            }
            c = prioSession.findTeamPriorities(
            new Long(iterationID));
            Iterator i = c.iterator();

            while(i.hasNext())
            {
                Prioritization ptemp =
                (Prioritization)i.next();
                prioritized = ptemp.getPrioritized();
            }

            if(taskID != null && prioritized)
            {
                ProcessSession pSession =
                EJBRemoteHomeCache.getInstance().
                getProcessSessionHome().create();
                Collection subProcessC = pSession.
                getSubprocesses(new Long(iterationID));
                prioSession.moveTask(new Long(taskID),
                subProcessC, direction);
            }

        }
        catch (FinderException e)
        {
            e.printStackTrace();
        }
        catch (CreateException e)
        {
            e.printStackTrace();
        }
```

```
        request.getRequestDispatcher("milosMain.jsp?side=/
        MilosXP/sideLinks.jsp&content=/MilosXP/
        whiteboard.jsp").forward(request, response);

    }
}
```

Listing 4: MoveTaskServlet and SaveFinalPrioritiesServlet

```
public void testAddAndRemoveTasks() { try {
    prioSession.addEmptyRows(proj.id, iterationID, taskID1);
    prioSession.addTask(iterationID, manager.id, taskID2);
    prioSession.addTask(iterationID, new Long(0), taskID2);

    prioSession.removeTask(taskID1);
    prioSession.removeTask(taskID2);

} catch (CreateException e) {
    fail("CreateException testing AddAndRemoveTasks: " +
    e.getMessage());
} catch (NamingException e) {
    fail("NamingException testing AddAndRemoveTasks: " +
    e.getMessage());
} catch (RemoteException e) {
    fail("RemoteException testing AddAndRemoveTasks: " +
    e.getMessage());
} catch (FinderException e) {
    fail("FinderException testing AddAndRemoveTasks: " +
    e.getMessage());
} catch (RemoveException e) {
    fail("RemoveException testing AddAndRemoveTasks: " +
    e.getMessage());
}
}
```

Listing 5: test function

References

[Amb01] Scott W. Ambler. *Agile Modeling*. John Wiley & Sons., 2001.

[Bec99] Kent Beck. *Extreme Programming explained*. Addison-Wesley, 1999.

[Coc02] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2002.

[CS99] Peter Checkland and Jim Scholes. *Soft Systems Methodology in Action*. John Wiley & Sons., 1999.

[Dav94] Alan M. Davis. *Software Requirements Revision Objects, Functions, & States*. Prentice Hall PTR, 1994.

[E2S03] HOORA E2S. *Use cases*. 2003.

[ERJ02] Scott Ambler Ed Roman and Tyler Jewell. *Mastering EJB*. John Wiley & Sons., 2002.

[Fow00] Martin Fowler. *The new methodology*. http://www.martinfowler.com/articles/newMethodology.html, 2000.

[Gro95] Standish Group. *Chaos Report*. www.standishgroup.com, 1995.

[Hig96] James A. Highsmith. *Adaptive Software Development*. Dorset House Publishing, 1996.

[KS97] Gerald Kotonya and Ian Sommerville. *Requirements Engineering*. John Wiley & Sons, 1997.

[LL01] O. López and M.A. Laguna. Requirements reuse for software development. In *Fifth IEEE International Symposium on Requirements Engineering*, Toronto, Canada, August 2001.

[Mac96] Linda A. Macaulay. *Requirements Engineering*. Springer Verlag, 1996.

[McP01] Christopher McPhee. Requirements engineering for projects with critical time-to-market. Master's thesis, University of Calgary, 2001.

[Nie97] Jakob Nielsen. *The Use and Misuse of Focus Groups.* http://www.useit.com/papers/focusgroups.html, 1997.

[PAW02] Jussi Rankainen Pekka Abrahamsson, Outi Salo and Juhani Warsta. *Agile software development methods - Review and analysis.* VTT Electronics, 2002.

[PCL99] Eric Lefebvre Peter Coad and Jeff De Luca. *Java Modeling in Color with UML.* Prentice Hall PTR, 1999.

[Pot98] Colin Potts. *Requirements Reviews: Understanding Customer Requirements.* http://www.cc.gatech.edu/ computing/classes/cs3302_98_summer/7-21-reqts/sld001.htm, 1998.

[Rob00] Suzanne Robertson. Requirements testing: Creating an effective feedback loop. In *FEAST 2000*, London, July 2000.

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum.* Prentice Hall PTR, 2001.

[Sta95] Jennifer Stapleton. *DSDM - Dynamic System Development Method.* Addison-Wesley, 1995.

[Wie99] Karl E. Wiegers. *Software Requirements.* Microsoft Press, 1999.