User-Centered Design in Agile Software Development

Pirkka Rannikko

University of Tampere
School of Information Sciences
Interactive Technology
M.Sc. Thesis
Supervisors: Saila Ovaska and Timo Poranen
April 2011
There are issues in traditional software development, e.g., the lack of user input and incomplete and changing requirements that can cause problems in projects. Both User-Centered Design (UCD) and agile software development are iterative approaches to software development, and they can increase the chances for delivering a successful project. UCD is a design process focusing on user research, user interface design and usability evaluation. Agile software development refers to a number of iterative and incremental software development methodologies that emphasize people, communication and the ability to adapt to change.

Integrating UCD and agile development would create a comprehensive systems development methodology. They do not have inherent obstacles that could prevent such integration, but both need to change before it can succeed. While no 'unified' Agile UCD methodology has been established, there has been a growing interest in learning how to integrate these two proven approaches over the last few years. In this thesis we explore the integration of UCD and agile development through experience reports and studies that have been published in the industry and academia. We sketch a complete picture of the integrated development process and the related best practices. For example, design chunking and parallel track development are needed to fit UCD work into the fast paced life-cycle of agile development. In addition we provide some advice related to the organizational adoption of the process and its practices.

**Key words and terms:** agile software development, interaction design, software project management, UCD, usability, usability engineering, user-centered design, user experience, user interface design
# Contents

1. Introduction ................................................................................................................... 1
2. Issues in traditional software development ................................................................. 3
3. User-Centered Design .................................................................................................. 8
   3.1. UCD background ................................................................................................. 8
   3.2. UCD life-cycle .................................................................................................... 9
   3.3. UCD practices and methods ............................................................................. 15
   3.4. UCD benefits and criticism ............................................................................. 17
4. Agile software development .......................................................................................... 20
   4.1. Agile background ............................................................................................... 20
   4.2. Different methodologies .................................................................................... 22
       4.2.1. Extreme Programming (XP) ................................................................... 23
       4.2.2. Scrum ....................................................................................................... 24
   4.3. Agile life-cycle .................................................................................................... 25
   4.4. Agile practices and methods ............................................................................. 29
   4.5. Agile benefits and criticism ............................................................................. 33
5. Similarities and differences in UCD and agile development ........................................... 37
   5.1. The challenge of agile UCD ............................................................................. 37
   5.2. Iterative process ............................................................................................... 38
   5.3. User vs. customer as a stakeholder ................................................................. 38
   5.4. Generalist vs. specialist: Role of the UCD specialist in a team ....................... 40
   5.5. Up-front design ............................................................................................... 41
   5.6. Testing ............................................................................................................. 43
6. Integrating UCD into agile development ..................................................................... 45
   6.1. Integration principles ....................................................................................... 45
   6.2. Integrated life-cycle ........................................................................................ 47
   6.3. Project management of UCD work in the integrated process ......................... 52
   6.4. Best practices to agile UCD ............................................................................. 53
7. How to adopt agile UCD in an organization ................................................................ 55
   7.1. Incentives for process improvement ............................................................... 55
   7.2. Considerations regarding agile adoption ....................................................... 57
   7.3. Considerations regarding UCD adoption ........................................................ 59
8. Conclusions .................................................................................................................. 65
References .......................................................................................................................... 67
1. **Introduction**

Since I was first introduced to software development, I've been both fascinated about it and puzzled how it should be done. Who are the people involved, what work should be included, how should it be organized and what kind of a process is it when all are put together? What is the big picture?

Later on I learned that, if software development projects are organized and executed like traditional engineering projects, problems will follow: poor communication, ignoring change and not involving end-users will result in problems when it comes to scope, schedule and budget of a project. The product might be late, too expensive, of low quality and not work properly. It might not even meet the users' needs being an entirely wrong product for its intended market segment or user group. Fortunately the industry has learned also and there are other ways of doing software development that better serve its intangible nature.

User-Centered Design (UCD) and agile software development (agile) are iterative approaches to software development that can increase the chances that a project will be a success. UCD does this by taking the product's end-users into account by researching them and involving them in the iterative design and evaluation of the product. This will result in a product that is more usable and that meets the users' needs better than a product developed without such emphasis on users. While UCD concentrates on learning about the would-be users and their needs and requirements in product development, agile software development concentrates on actually crafting the software and to the necessary project management. There is no single unified agile methodology and the term is used to refer to a number of different iterative & incremental methodologies that emphasize people, communication and the ability to adapt to change rather than the process, tools and predictive planning.

While both approaches are potentially beneficial to software development they alone don't cover how to develop *usable, working software within the project's constraints*. UCD only deals with the user research and the design and evaluation of the user interfaces of the product. Agile methodologies on the other hand seem to forget end-users and usability altogether as Larry Constantine (2002), one of the developers of the usage-centered design process, has observed. While both approaches have similarities, some of their aspects can be seen to conflict each other (Blomkvist, 2005) possibly preventing the creation of an integrated systems development process.

Early literature from both camps seem to ignore the other for the most parts, but over the last few years there has been an increasing interest in learning how to integrate these two proven approaches. For instance, in the Agile conference series there has been a whole stream dedicated to discussion about agile user experience since 2008. Despite
the increased discussion, experience reports and a few studies, there isn't a wealth of refined information waiting to be taken into practice. A complete picture of the integrated agile UCD process is still waiting to be painted.

The purpose of this thesis is to sketch that picture by discussing what UCD and agile are and how they can be integrated and used to improve the quality of software products. Studying software development in the real world is problematic. Each organization, project and a team is unique and comparing their way of working with each other is challenging. Even if such comparison could be made, the nature of software development make it even more unpredictable, e.g., a programmer could turn out to be ten times as productive as the colleague next to him. And it is the people who craft the software, not the development processes.

Processes are needed to organize the people and the work they do to achieve a common goal. Thus the main focus of this thesis is the integration of UCD and agile development and the process life-cycle. The subject is explored from project management and organizational adoption perspectives but process improvement metrics and measuring its success are not discussed. In other words my goal is to find answers to the following questions:

- What are the key UCD and agile techniques that have proven to be effective in the industry?
- How have UCD and agile been integrated into a single comprehensive software development process?
- What kind of a life-cycle this agile UCD process would have?
- How should agile UCD be adopted in an organization?

The thesis is divided into three larger themes. In the first theme some of the problems of traditional software development are introduced to give an idea why UCD and agile development are needed in the first place. These approaches are also introduced to give enough background for the second theme which is the discussion about the integration of UCD into agile development. This theory is mainly based on material published by the pioneering agile UCD practitioners in the software development industry. The third theme is organizational adoption or how agile development and UCD can be adopted in a software development organization. After reading this thesis anyone new to software development will get an introduction to some of its issues and possible answers how to solve them. UCD and agile practitioners on the other hand can hopefully pick some concrete advice and apply it in their work in order to create better software.
2. Issues in traditional software development

Before we can discuss software development projects (and what there is to improve) we need to consider what is a project in general terms. According to the Project Management Body of Knowledge or PMBOK Guide “a project is a temporary endeavor undertaken to create a unique product, service, or result” (PMBOK Guide, 2004, p. 5). Each project has a definite beginning and an end which is reached when the project's objectives have been met or the project is terminated. Operations or 'normal' work in contrast is ongoing and repetitive and adapts to a new set of objectives to sustain business (PMBOK Guide, 2004, p. 7). Projects are constrained by their scope, time and cost, and a change in one will affect the others. A successful project is delivered within these constraints.

Project management on the other hand can be defined as “the application of knowledge, skills, tools and techniques to project activities to meet project requirements” (PMBOK Guide, 2004, p. 8) or, in layman's terms, activities that are needed to organize the work. A project life-cycle can be divided into initiation, execution and completion phases where the execution phase is divided further into activities such as discovery, design, development and evaluation of the project's end-product. These activities are formed from a number of specific tasks.

Similarly, software development projects usually include four types of work activities in the execution phase (in addition to project management) that are needed to deliver any non-trivial system:

- research (requirements engineering),
- design (software engineering or technical design),
- programming, and
- testing.

In a traditional software development process these activities are executed in a sequence of separate steps where ideally each step is completed before the next one begins. The idea is to first specify the requirements (what will be built) completely and then design or engineer (how it will be built) the software in detail. These steps are sometimes referred to as the big design up front (Sliger & Broderick, 2008, p. 321) principle and the results are usually documented in detail in deliverables like a vision & scope document, a requirements/functional specification and architecture/design documentation. After requirements and design, the software will be programmed and finally tested.

An alternative to the sequential project life-cycle model is the iterative and incremental model. In this model the overall life-cycle is composed of several iterations, each being a self-contained mini project where all the work activities (requirements,
design, development and testing) are performed concurrently. The goal of an iteration is to release an integrated, tested and partially complete but working system, which is then validated and improved in the next iteration. The system is built incrementally by adding new features and functionality over several iterations rather than having all of them in place after the first iteration. (Larman, 2003, pp. 9-10)

The sequential development process is often called the waterfall model. It was defined (Royce, 1970) to be an example (see Figure 1) suitable only for the most straightforward software projects. Ironically due its simplicity, general misconception and various other reasons it was adopted as the standard model for software development in the industry (Larman, 2003, pp. 102-106).

Although the waterfall type process works well in traditional engineering and predictable manufacturing, there is a lot of evidence (Cohn, 2005, p. 11; Larman, 2003, pp. 74-75) that it isn't very well suited for software development. In fact predictable manufacturing is an entirely wrong paradigm for software which should be considered as new product development instead (Larman, 2003, p. 4). Programming is not engineering and manufacturing. It is a craft. The failure to acknowledge these fundamentals is the root cause of many challenges is software development projects. According to the initial CHAOS report from Standish Group, only 16% of software projects were completed on-time and on budget (The CHAOS report, 1995). The three main factors that were reported to cause a project to be challenged were:

1. lack of user input,
2. incomplete requirements & specifications, and
3. changing requirements & specifications.
Software development is an abstract and uncertain business. Aguanno (2004) reports various reasons for uncertainty that will cause high level of change in projects. He writes that “at the early design stages, the intangible nature of most software leads to difficulties in communicating design and vision in an easily-understandable way”. Software products are usually unique with no available analogues to compare with, and progress is hard to estimate due to the intangibility. Also the tools and building blocks for creating software along with the industry standards are constantly changing. (Aguanno, 2004, pp. 20-21)

In the waterfall approach executable programs are often produced at the last possible moment. This is a problem as there is no user interface to test with real users and the normal work products can't be used as prototypes (Nielsen, 1993, p. 94). Abstract specification documents are much harder to understand than concrete prototypes. With them, it is hard to collect enough feedback from the users and other non-technical stakeholders early on in the project, when it would be needed the most. Feedback starts pouring in when the actual user interface or working application is seen the first time. If this happens late in the project, it might result in considerable changes in the product which in turn will cause problems scope, schedule and budget issues. Risks are uncovered late in the sequential development model.

The traditional approach to changes in software development is to lock down the requirements to form a baseline for the product, implement change control and use the documented requirements or contract to enforce the change control (Aguanno, 2004, p. 18). Larman (2003, p. 55) writes that attempts to face the challenge of changing requirements by early detailed requirements analysis and freeze rarely succeed.

Early attempts at predictive planning are more successful at repetitive manufacturing project of low change and complexity, but not on inventive projects – where the full requirements and risks are not reliably known at the start and high rates of change are the norm. (Larman, 2003, p. 61)

Requirements tend to change often in a software project, and predicting the future state of a long development project is nearly impossible. According to Cohn (2005, p. 4) Boehm has studied that a project schedule estimate can vary as much as 60%-160% of the actual schedule during the initial stages of a waterfall project and 85%-115% after the requirements specification. Despite of this uncertainty, contracts often have a fixed price and scope. The waterfall model was supposed to fix the problem of changing requirements, but the industry has found out that they could not be pinned down in a single iteration as had been anticipated (Aguanno, 2004, p. 55). Larman (2003, p. 55) cites that requirements can change 25% or more in a software project. Also failing to involve users in requirements analysis can cause considerable development efforts to be wasted when unnecessary features are developed as the real users' needs are not known.
Nielsen (1993, p. 5) cites a study where 63% of large software projects significantly overran their estimates and that the four most significant causes related to these failings were “frequent requests for changes by users, overlooked tasks, users' lack of understanding of their own requirements, and insufficient user-analyst communication and understanding”.

The traditional (and artificial) change control creates a false sense of predictability and projects are then planned in detail in to far future. This causes only problems as the planning is based on unreliable or false knowledge about the project and the product.

A fourth shortcoming with traditional approaches to planning is the failure to acknowledge uncertainty. We ignore uncertainty about the product and assume that the initial requirements analysis led to a complete and perfect specification of the product. We assume that users will not change their minds, refine their opinions, or come up with new needs during the period covered by the plan. (Cohn, 2005, p. 17)

The waterfall life-cycle pushes many high-risk and difficult elements like programming and testing the core architecture and interfaces towards the end of a project. It has also been called the *fail-late* life-cycle as there can be the illusion of accurate scheduling during the early easier phases of the project, but when the complex issues emerge later, the schedule breaks apart. (Larman, 2003, pp. 58, 62)

Mayhew (1999) raises another source of problems in software projects that relates to the sequential life-cycle and document-centric process model of the traditional software development approaches. Traditionally “different kinds of specialists perform different steps in the software engineering process, and communication between the steps/specialists is accomplished through documents” (Mayhew, 1999, p. 21). Writing, reading and understanding documents is very time consuming and always prone to misconceptions. Documentation is almost always incomplete and information is missing. Reading has also a very low 'bandwidth' when it comes to delivering information as opposed to face-to-face communication for example (Aguanno, 2004, p. 42). It is also a problem that the work of some specialists does not impact the work of those working before them in the earlier steps. Those coming late into the project might have little commitment to the work done before them. Based on my own work experience I can also say that a lot of tacit and valuable information about the project and the product is lost when the specialists leave the project after completing their 'own step'.

To sum up, there are a lot of problems in traditional software development approaches. Poor communication, predictive planning, ignoring uncertainty and change, not involving end-users, and pushing high-risk tasks until the end of the project will result in problems of project scope, schedule and budget. The product might be late, too
expensive, or of low quality, or it might not meet its users' needs at all, being an entirely wrong product for the intended market segment or user group. Time-to-market and the quality of user experience are factors that can mean life or death to products and companies. Thus, better ways of designing and building software are needed to satisfy the needs of the fast-paced industry. This is where User-Centered Design and agile software development step in.
3. User-Centered Design

3.1. UCD background

User-Centered Design is an approach to product development that puts emphasis on the end-users of the product by conducting user-research and involving them in the iterative design and evaluation of the product. The goal of UCD is to produce usable products that meet the users’ needs, but it does not only represent the techniques, processes, methods and procedures for achieving that, but also is “a philosophy that places the user at the center of the process” (Rubin & Chisnell, 2008, p. 12). UCD is not a complete methodology for software development when considering the whole life-cycle and activities of a project. It is meant to be integrated into and work as a sub-process for any existing software development methodology.

UCD has its roots in human factors engineering that began to emerge as a discipline after the World War I as human limitations and capabilities towards flying were studied and taken into consideration in aircraft cockpit design (Meister, 1999). Although the term user-centered design originates from Donald Norman’s work in the 1980s (Abras, Maloney-Krichmar, & Preece, 2004), many have contributed to its development (see Gould (1988) for example).

In the literature the term usability engineering (Nielsen, 1993) is used to describe more or less the same process and activities as in UCD or in User-Centered Systems Design (UCSD). Therefore I consider them interchangeable for the purposes of this thesis. There are a number of different practical design approaches like the Cooperative Design, Participatory Design and Contextual Design (Göransson, 2004, p. 36) but they all fall under the general category of UCD despite their different flavors. Interaction design could be seen as a relabeling of UCD (Göransson, 2004, p. 39) with more emphasis on aspects like graphical and industrial design. Goal-Directed Design (Cooper, Reimann, & Cronin, 2007) and Usage-Centered Design (Constantine & Lockwood, 1999) are practical approaches to interaction design.

User experience (UX) on the other hand is a different and much wider concept than usability and it “encompasses all aspects of the end-user's interaction with the company, its services, and its products” (Nielsen Norman Group, 2009). Although there is a difference between the concepts of UX and usability, in practice the term user experience is often used and has become quite a buzzword in the corporate world. UCD covers both of these concepts.

UCD largely deals with activities related to improving usability (or user experience). The ISO 9241-11 standard defines usability as: “The extent to which a product can be used by specified users to achieve specified goals with effectiveness,
efficiency and satisfaction in a specified context of use” (ISO 9241-11, 1998). Usability can also be understood as the 'absence of frustration' in using a product, and it is invisible in a way that only the lack of it will be noticed and cause dissatisfaction and complaints (Rubin & Chisnell, 2008, pp. 4, 6). Nielsen (1993, pp. 24-26) defines usability as a sub-attribute of usefulness (which is a part of the acceptability of a system) and divides it into five different attributes that apply to a system:

- learnability (easy to learn),
- efficiency (efficient and productive to use),
- memorability (easy to remember what has been learned),
- errors (users make few errors, good error handling and recovery from them),
- satisfaction (subjectively pleasing).

According to the ISO 9241-11 standard the “usability of products can be improved by incorporating features and attributes known to benefit the users in a particular context of use” (ISO 9241-11, 1998). As with all quality attributes, usability is not a separate feature or something that can be added to a product at the end like a cream on top of a cake. The usability of a product is the result of systematic UCD work that starts at the beginning of the project and continues even after the product has been released with work towards its future versions.

3.2. UCD life-cycle

UCD work is not a one-time activity where the user interface is fixed before the release of a product, but a set of activities that are carried out throughout the life-cycle of the product's development. There is a need for multiple stages of UCD work that supplement each other in the project. Usability cannot be seen in isolation from the broader corporate product development context. Usability efforts made now may reap even greater return on investment in the future as new releases in the product family benefit from those efforts. (Nielsen, 1993, p. 71)

Although there are many specific UCD methods and practices for different purposes, the process generally includes iterative activities for research, design and evaluation in which the actual end-users participate. The ISO 13407 standard documents one approach (Human-centred design processes for interactive systems) to UCD. It is meant to be a complementary method to any existing software development methodologies. It is characterized by the active involvement and a clear understanding of users and task requirements, an appropriate allocation of function between users and technology, the iteration of design solutions and multi-disciplinary design (ISO 13407, 1999, p. 3). The standard also specifies four human-centered activities that should be iterated during a project (ISO 13407, 1999, p. 5):

- to understand and specify the context of use,
- to specify the user and organizational requirements,
• to produce design solutions, and
• to evaluate design against requirements.

Gulliksen, Göransson, Boivie, Blomkvist, Persson, & Cajander (2003) argue that the UCD principles listed in the ISO 13047 standard are not enough to maintain UCD in a project or in an organization and that the UCD process needs to be specified in more detail to avoid problems. They have drawn together 12 key principles for UCD (Table 1) and present an evolutionary life-cycle model based on the principles (Figure 2). (Gulliksen et al., 2003)

Table 1: Key principles for User-Centered Design (Gulliksen et al., 2003).

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User focus</td>
<td>The goals of the activity, the work domain or context of use, the users’ goals, tasks and needs should guide the development early.</td>
</tr>
<tr>
<td>Active user involvement</td>
<td>Representative users should actively participate, early and continuously throughout the entire development process and throughout the system life-cycle.</td>
</tr>
<tr>
<td>Evolutionary systems development</td>
<td>The systems development should be both iterative and incremental.</td>
</tr>
<tr>
<td>Simple design representations</td>
<td>The design must be represented in such ways that it can be easily understood by users and all other stakeholders.</td>
</tr>
<tr>
<td>Prototyping</td>
<td>Early and continuously, prototypes should be used to visualize and evaluate ideas and design solutions in cooperation with the end users.</td>
</tr>
<tr>
<td>Evaluate use in context</td>
<td>Baselined usability goals and design criteria should control the development.</td>
</tr>
<tr>
<td>Explicit and conscious design activities</td>
<td>The development process should contain dedicated design activities.</td>
</tr>
<tr>
<td>Professional attitude</td>
<td>The development process should be performed by effective multidisciplinary teams.</td>
</tr>
<tr>
<td>Usability champion</td>
<td>Usability experts should be involved early and continuously throughout the development life-cycle. (Note: Usability champion is a senior designer in the project with the mandate to decide on usability matters and the ability to mentor others. It differs from the definition of Rubin &amp; Chisnell, see Section 7.3.).</td>
</tr>
<tr>
<td>Holistic design</td>
<td>All aspects that influence the future use situation should be developed in parallel.</td>
</tr>
<tr>
<td>Process customization</td>
<td>The UCD process must be specified, adapted and/or implemented locally in each organization.</td>
</tr>
<tr>
<td>A user-centred attitude should always be established</td>
<td>All people involved must be aware of and committed to the importance of usability and user involvement.</td>
</tr>
</tbody>
</table>
Nielsen (1993, p. 72) describes the UCD process as a collection of tasks from which at least some are repeated multiple times making the process iterative. He notes that even if the project does not include every task from the UCD process, the effort to improve usability can still be successful. The tasks in Nielsen's model are:

- know the user,
- competitive analysis,
- setting usability goals,
- parallel design,
- participatory design,
- coordinated design of the total interface,
- apply guidelines and heuristic analysis,
- prototyping,
- empirical testing,
- iterative design,
- collect feedback from field use.

One of the most important aspects of UCD is its iterative nature. In Nielsen's life-cycle model the design and testing of the user interface are done in multiple iterations gradually improving and refining the final product. If the user interface were designed
in a single iteration and tested only in the end in a waterfall type project, there wouldn't be time left to improve the product (based on the feedback collected from the users) before the release of the product. Even if there were time to make changes before the release, their cost could be way out of the budget. The cost of changes in software development increases exponentially as the project moves from requirements to design to development and finally to testing. The most serious and often underlying problems would not get fixed and the changes made at this point could end up being just superficial polishing. Dealing with the discovered issues is cheaper by multiple orders of magnitude early in the project when the team is working with quick throw-away prototypes. The team should plan enough iterations (at least three) as the user interface might actually get worse after the first iteration before it gets any better (Nielsen, 1993, p. 107).

But as said earlier, UCD is not a complete process model for software development as it only deals with user research, design and usability testing and evaluation. It needs to be integrated into a software development methodology so that they together cover the whole life-cycle of the project. For example, if the four human-centered activities of the ISO 13407 standard are integrated into a traditional waterfall project they would be performed iteratively and in parallel with the normal software requirements engineering and design. Figure 3 presents the four human centered activities in the context of Royce's model.

Figure 3: UCD in a waterfall project. Modified after Royce (1970) and ISO 13047 (1999).
Ideally the UCD work should be an integral part of the requirements engineering activities or at least done in close collaboration to ensure that end-users' needs are really taken into consideration. User research would have a bigger emphasis in the beginning but the focus would gradually shift to producing and validating design solutions in further iterations. The UCD and requirements work would be completed before development. The testing step in Figure 3 could also include a final verification usability test (Rubin & Chisnell, 2008, p. 35) (along the other acceptance tests) to measure how the complete system meets the usability goals that were set.

Mayhew (1999) has also defined a complete integrated model for UCD that embeds into and is performed parallel to the existing software development process (see Figure 4 for an overview). The model is divided into requirements analysis, design/testing/development and installation life-cycle phases. The user interface design and evaluation is done iteratively on three different levels of detail: concept & wireframes, screen design standards (which are comparable to a user interface style guide) and the detailed user interface. (Mayhew, 1999, pp. 6-9)

Mayhew (1999, p. 21) makes an important point that the “optimal implementation of the life-cycle requires full participation of cross-functional teams” (Mayhew, 1999, p. 21). Most of the UCD tasks should not be carried out in isolation by the UCD specialists but by the whole project team that the UCD specialists are part of. All team members should carry out the life-cycle tasks jointly to develop a shared understanding of the requirements and design issues. Mayhew describes the cross-functional team in the following way:

All team members perform all steps together in the overall process, each bringing his or her particular perspective and skill set to all major project decisions. In this way communication is facilitated, and a deeper shared understanding of issues across team members results. In addition, all participating specialists get their issues addressed, as each step is performed to minimize rework downstream in the process. Finally team members who ultimately will make design decisions not only understand requirements issues but have bought into the goal of addressing them explicitly in the design. (Mayhew, 1999, p. 22)

Although the project teams should be multidisciplinary, the UCD staff should be centralized into a single team in the organization. Both principles, cross-functional team and centralized UCD staff, are beneficial to UCD in general (for example, to its adoption, see Section 7.3.) and also to its effectiveness (Vredenburg, Mao, Smith, & Carey, 2002).
Figure 4: Mayhew’s usability engineering life-cycle (Mayhew, 1999).
3.3. UCD practices and methods

Work in the UCD life-cycle in general consists of user research and the iterative design and evaluation of the product (and of course the planning of the work). These activities include specific tasks like setting up usability goals, designing the information architecture of a website or conducting a usability test on a user interface. A number of specified techniques can be used to accomplish different UCD activities.

In this section I'll summarize those UCD activities and techniques that are relevant to software development. They have been collected from various sources (Mayhew, 1999; Nielsen, 1993; Ovaska, Aula, & Majaranta, 2005; UsabilityNet: Methods, 2006) and are presented in Table 2. The table is divided into sections according to the five general activities performed in software development projects (project management, research, design, programming and testing). Each section (except programming) includes a number of UCD activities and tasks which are grouped in to a column named Practice. Each of these practices has a short description, list of different techniques that can be used and any specific work product they produce besides standard written documentation.

Table 2: UCD practices and methods.

<table>
<thead>
<tr>
<th>Practice</th>
<th>Description</th>
<th>Techniques</th>
<th>Work products</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project management</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cross-functional team</td>
<td>The whole team participates in the UCD activities to develop a shared understanding of requirements and design issues and to make a commitment to the usability of the product.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Research</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User research</td>
<td>The purpose of user research is to discover who the actual users are, what goals they have and what is the context where the product is used. The information is used to direct the design and development of the product to match the users' needs.</td>
<td>Surveys, Interviews, Observation, Site visit, Diary, Stakeholder interview, Focus group, Contextual inquiry, Affinity diagram, Cultural probe</td>
<td>User profiles, Personas, User scenarios</td>
</tr>
<tr>
<td>Task analysis</td>
<td>Task analysis takes user research a step further by discovering and analysing specific tasks the users perform with the product to reach their goals. The information is used to design new or to improve existing workflows to match the users' needs.</td>
<td>Flowcharts, Diagrams, Sitemaps, User scenarios, Use cases</td>
<td></td>
</tr>
<tr>
<td>Competitor analysis</td>
<td>Competitor analysis is not only a tool for the sales or marketing. It can be used to learn how well features of other products support those tasks the planned new product is expected to have.</td>
<td>Expert evaluation, User testing</td>
<td></td>
</tr>
<tr>
<td>Setting usability goals</td>
<td>The product should have an agreed set of qualitative and quantitative usability goals that can be used to measure if the product is 'usable enough'. Also a financial impact analysis should be made to calculate the monetary value and the return on investment for the UCD activities to be performed.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Design

| Concept design | Design at the conceptual level usually includes such aspects as the information architecture of a system, user workflows & tasks and low-fidelity wireframes to model the user interface at low level of detail. Concept design can also include redesign of whole business strategies, processes and services. Although design at that level has less to do with the actual UI it still falls under the umbrella of concept and service design. **• Participatory design** **• Parallel design** **• Design Studio** **• Card sorting** **• Prototyping** **• Design patterns & guidelines** **• Flowcharts** **• Diagrams** **• Sitemaps** **• User scenarios** **• Use cases** **• Wireframes**

| User interface design | Design at the detailed user interface level involves the drawing of high-fidelity UI mockups to model and describe the UI in detail. Usually design patterns and platform UI design guidelines are used to guide the design. Functional prototypes can be built for analysing the design solutions in practice. The design principles could be documented in a UI style guide to guide the future development efforts. **• Wireframes** **• Mockup images** **• Functional prototypes** **• Style guide**

### Programming

| - | - | - | - | - |

### Testing

| Expert evaluation | The goal in an expert evaluation is to discover usability problems through a review of the product conducted by a user interface designer or some other qualified specialist. Usually the review is made by evaluating the product against a list of design principles or heuristics. **• Heuristic evaluation** **• Cognitive walkthrough** **• Usability walkthrough** **• Evaluation report**

| Usability testing | The purpose of a usability test is to discover usability problems by testing the user interface (prototype or complete) with actual user representatives carrying out specific tasks with the product. The subjective user experience of the test participants is usually also measured with interviews after the tasks have been completed. The test results should be measured against the usability goals to see if they’ve been met. **• Interviews** **• Observation** **• Think-out-loud** **• Wizard of Oz** **• Eye tracking** **• Formative test** **• Summative test** **• Verification test** **• Remote testing** **• Test report**

As shown in the table, the UCD practices concentrate on research, producing design solutions and testing those solutions. Practices related to the actual building of software or even project management activities are almost non-existent.

In most cases it is not feasible to perform every possible UCD task in a project. Nielsen (1993, pp. 16, 112) suggests that under tight budget and time constraints the most important discount usability engineering techniques should be used. These include site visits for making user and task observations, prototyping with user scenarios, usability testing with simplified thinking aloud, and heuristic evaluation.

According to one study (Vredenburg et al., 2002) the five commonly used UCD methods are iterative design, usability evaluation, task analysis, informal expert review and field studies. All of these methods were also considered to be the most effective except informal expert review. Also user requirements analysis was considered to be an effective method but it was seldom used. It seems that these two methods have a strong cost-benefit tradeoff.
3.4. UCD benefits and criticism

Using the different UCD methods and practices throughout the project's entire life-cycle will increase the usability of products, but what are the actual benefits of a more usable product? The ISO 13407 standard states that “Making interactive systems more human-centered has substantial economic and social benefits” (ISO 13407, 1999, p. 2) and that usable systems

- are easier to understand and use, thus reducing training and support costs,
- improve user satisfaction and reduce discomfort and stress,
- improve the productivity of users and the operational efficiency of organizations, and
- improve product quality, appeal to the users and can provide a competitive advantage.

Marcus (2005, pp. 20-22) also lists several benefits (in the form of savings and increases) related to UCD and its use in product development (Table 3).

<table>
<thead>
<tr>
<th>Savings</th>
<th>Increases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save development costs</td>
<td>Increase transactions / purchases</td>
</tr>
<tr>
<td>Save development time</td>
<td>Increase product sales</td>
</tr>
<tr>
<td>Reduce maintenance costs</td>
<td>Increase traffic, size of audience</td>
</tr>
<tr>
<td>Save redesign costs</td>
<td>Retain customers</td>
</tr>
<tr>
<td>Decrease support costs</td>
<td>Attract more customers</td>
</tr>
<tr>
<td>Reduce training/ documentation cost</td>
<td>Increase market share</td>
</tr>
<tr>
<td>Avoid litigation (by taking care of product safety)</td>
<td>Increase success rate, reduce user error</td>
</tr>
<tr>
<td></td>
<td>Increase efficiency / productivity</td>
</tr>
<tr>
<td></td>
<td>Increase user satisfaction</td>
</tr>
<tr>
<td></td>
<td>Increase job satisfaction / decrease job turnover</td>
</tr>
<tr>
<td></td>
<td>Increase ease of use</td>
</tr>
<tr>
<td></td>
<td>Increase ease of learning</td>
</tr>
<tr>
<td></td>
<td>Increase trust in systems</td>
</tr>
</tbody>
</table>

Based on these benefits we can see that for organizations, investing in UCD activities in product development generally pays back in two different ways: 1. It brings cost savings due to increased productivity and decreased need for customer/user support. 2. It produces increased monetary income due to increased number of sales and leads generated. Benefits of UCD efforts also apply to in-house development; Nielsen (2007) has estimated that productivity gains of 2-3 million US dollars per year could be
reached in an organization that has 10,000 employees by increasing the usability of its intranet from “poor to average”.

The expected return on investment (ROI, the income your investment pays back) for usability activities should always be estimated at the beginning of the project, and calculated later to make a case for the cost justification of those activities and to promote the benefits of UCD further in the organization. It is not easy though. Usually other variables such as a new pricing plan or the new features of the product make it even harder. Rosenberg (2004, pp. 22-29) has presented some criticism against making ROI analysis, suggesting that it does not meet the real needs of the software industry at enterprise scale. Nonetheless, it is still worthwhile to try to do the calculations as the numbers can be quite convincing in selling usability.

Nielsen has conducted studies on usability budgets and ROI. Based on the data collected from 863 different design projects, he recommends that about 10% of the project's budget should be used on usability efforts. In 2003 this average investment generated an increase of 135% in the key performance indicators usually related to websites such as conversion rates, traffic numbers, user performance and target feature usage (Nielsen, 2003). In 2008 the corresponding increase was 83% (Nielsen, 2008).

Marcus (2005) cites case studies where organizations have hugely benefited from UCD. For example after the redesign and re-launch of the IBM online store in 1999 their traffic increased 120% and sales went up 400% after the first month (Marcus, 2005, p. 28). Another example is a company that invested 20,700$ on a usability effort to improve a sign-on procedure in a system with thousands of users. The investment brought cost savings of 41,700$ on the first day the system was used (Marcus, 2005, p. 30).

There isn't much critique presented against UCD, but the most common arguments are that it can cost a lot and take a long time to do slowing down the development process (Abras et al., 2004). One of the great usability myths is that “usability is just common sense” (Bias & Karat, 2005, p. 10). Organizations might be reluctant to invest in dedicated usability efforts as they believe that the designers and developers will end up producing solutions that match users' needs without any extra effort. Of course this could not be further from the truth; the world is filled with products and software with poor usability. It is true that UCD activities take their time and money, but the benefits can hugely out-weigh the costs (actually saving development costs and shortening the time to market). It should also be remembered that even under major schedule pressures, it is better to perform some UCD activities than leave them out altogether.

One of the real problems relates to the notion of UCD. Since it is vaguely defined it can be applied in a variety of ways. This may lead to poor quality and poor usability
of the product and misconceptions about the effectiveness of UCD (Gulliksen et al., 2003). Although the ISO 13407 standard defines the UCD process, it is too abstract to be integrated into an existing software development process as such. Constantine and Lockwood (2002) write that although the three main UCD techniques (user studies, rapid prototyping and usability testing) are useful, they still are not substitutes for good design. They state that user studies easily confuse what users want with what they really need, that rapid iterative prototyping is often a sloppy substitute for systematic design and that usability testing is often an inefficient way to find problems that could have been avoided through proper design.

Personally I can't see a reason why a systematic model-based design approach, like the Usage-Centered Design (Constantine & Lockwood, 1999), couldn't be included in the UCD process. The trick is to choose and plan those UCD practices that are suitable for the project in the first place and which yield the greatest benefits while being cost-effective. Usability is always tied to the context of use and the users, and even in an organization where a defined UCD process is in place, the planning of the UCD work must be done on a project to project basis.
4. Agile software development

4.1. Agile background

Agile software development is an approach to software development that, in addition to programming, concentrates on subjects like project management and team work. Agile is a philosophy or a way of thinking about software development and there is no single unified agile methodology to follow (Shore & Warden, 2008, p. 9). The term agile also refers to a number of different iterative and incremental software development methodologies that share common principles and practices. These methodologies emphasize people, communication and the ability to adapt to change rather than the process, tools and predictive planning. The methodologies “are processes that support the agile philosophy” (Shore & Warden, 2008, p. 9) and each of them consists of individual practices and techniques.

Many of the agile methodologies (then called as lightweight) were created in the 1990s (Sliger & Broderick, 2008, p. 13) as an alternative to the traditional sequential (waterfall), document-centric and often heavyweight software development processes and their problems. Although agile methodologies are relatively new, some of their concepts like iterative and incremental development (IID) can be traced back to the 1930s (Larman, 2003, p. 80). NASA has used IID in software projects since the 1960s and IBM from the 1970s (Larman, 2003, pp. 80-83) and it has been promoted by several software development thought leaders since the 1970s (Larman, 2003, pp. 93-99). Also the ideas of Lean Product Development (used and propagated by Toyota in automobile production) have influenced the development of agile methodologies (Sliger & Broderick, 2008, p. 12) as they spread to North America and to the IT community at large in the 1980s (Aguanno, 2004, p. 18).

The actual term agile software development was coined in 2001 when 17 lightweight methodologists got together and wrote the Agile Manifesto (Sliger & Broderick, 2008, pp. 13-14). The Manifesto encapsulates the agile philosophy into four values or statements as displayed in Figure 5.
Manifesto for Agile Software Development

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

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

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

Kent Beck  James Grenning  Robert C. Martin
Mike Beedle  Jim Highsmith  Steve Mellor
Arie van Bennekum  Andrew Hunt  Ken Schwaber
Alistair Cockburn  Ron Jeffries  Jeff Sutherland
Ward Cunningham  Jon Kern  Dave Thomas
Martin Fowler  Brian Marick

Figure 5: Manifesto for agile software development (2001).

The Manifesto is also accompanied by the following 12 principles that reflect its four values (Manifesto for agile software development, 2001):

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

Different agile software development methodologies share these values and principles and follow them in their unique practices (see Section 4.4.).
4.2. Different methodologies

All agile methodologies share many common practices like iterative and incremental development and delivery, adaptive planning and put open face-to-face communication and people before documentation, processes and tools (Larman, 2003, pp. 25, 30). They all embrace change by adapting to the situation rather than doing detailed predictive planning, locking down requirements and implementing change control. Aguanno (2004, p. 21) writes:

When change occurs, these methods provide ways of allowing change to be introduced into the project in an orderly way that attempts to maximize the benefits for the sponsor, while controlling the risks that the change introduces.

The key to this is to keep the feedback cycle short by working in short timeboxed (fixed length) iterations (see Section 4.3.). Collecting stakeholder feedback by displaying working software in every iteration reduces the risk of building a wrong product. (Aguanno, 2004, p. 23)

Besides working and delivering in short iterations, an agile team works as a one sharing a common goal (Cohn, 2005, p. 23). Problems are solved together regardless of roles. Documents are no longer passed from one specialist to another as the primary means of communication. Programming is seen as a comprehensive craft. Besides writing the code, it also includes the technical design (modeling) and testing of the code. Agile teams focus on business priorities by delivering complete user-valued features in customer specified order to optimize the ROI (Cohn, 2005, p. 25). Teams also include an on-site customer representative that works with the team daily to give feedback and define requirements for the software (Shore & Warden, 2008, p. 29). This interactive face-to-face communication (Cockburn, 2004, p. 99) (and other practices) allows the team to develop the software without needing a detailed written documentation (like a traditional software requirements specification). If these stakeholders are not available, much of the agility of the project is lost as requirements need to be collected and documented by traditional means.

Agile teams are self-organizing (Larman, 2003, p. 34). The role of the project manager is to lead, guide and coach the team rather than directly manage the tasks it is working on (Wells, 2009a). Although agile teams organize and manage their own work, agile development should not be considered undisciplined. On the contrary, it takes quite a lot of discipline and rigor to build software while following an agile process (Ambler, 2007).

Extreme Programming and Scrum (see Subsections 4.2.1. and 4.2.2.) are the most notable of the agile methodologies, but there are also several others like Adaptive Software Development (ASD), Agile Unified Process, Crystal Clear, Dynamic
Solutions Delivery Model (DSDM), Feature Driven Development (FDD) and Lean Software Development (Sliger & Broderick, 2008, pp. 295-300). Agile Modeling and Pragmatic Programming on the other hand are not complete development processes but a set of principles and practices that can be used with other methodologies (Larman, 2003, pp. 38-39). In addition, Wikipedia lists methodologies like Essential Unified Process and Open Unified Process that have been developed from the IBM Rational Unified Process (Agile software development, n.d.). Furthermore, a process guidance called Microsoft Solutions Framework for Agile Software Development (MSF, n.d.) was created especially for software development teams working with Microsoft technologies and the Visual Studio system.

4.2.1. Extreme Programming (XP)

XP was created by Kent Beck and others (Wells, 2009b) while working for Chrysler Corporation and was first published by Beck (1999). The name reflects the idea that teams should take good, proven engineering practices to the extreme (Sliger & Broderick, 2008, p. 296). XP stresses “customer satisfaction through rapid creation of high-value software, skilful and sustainable software development techniques and flexible response to change” (Larman, 2003, p. 139). XP project life-cycle is divided into 1-4 week iterations (preference on the shorter) and the teams are relatively small (5-20 members) (Shore & Warden, 2008, pp. 39, 247-248).

XP advocates four values: communication, simplicity, feedback and courage. It can be characterized by its five basic principles: rapid feedback, assuming simplicity, incremental change, embracing change and doing quality work. In addition to the values and principles, XP includes 12 existing software engineering practices (see Section 4.4 for descriptions) which it combines for greater synergy (Cohn, 2004, pp. 234-241):

- small releases,
- planning game,
- refactoring,
- testing,
- pair programming,
- sustainable pace,
- collective code ownership,
- coding standard,
- simple design,
- metaphor,
- continuous integration, and
- on-site customer.

The emphasis of XP’s practices are on programming and the quality of code, but it is also a communication and team oriented methodology. XP does not require other
detailed work products (like a requirements specification document) but program code and test cases. Oral communication is the suggested way of working with requirements and design. The whole team, including customers, developers and managers, is expected to work together in the same project space to quickly deliver software with high business value. (Larman, 2003, pp. 139-140)

The customer's role in an XP project is to document software requirements/features as user stories (a simple description of a feature/functionality), prioritize these stories by their business value and write and execute tests that demonstrate that the stories are implemented as expected. The XP programmer role is versatile making no distinction between programmers, designers, testers and so on. All programmers work as a team and share responsibilities that might be assigned to specific individuals in a non-XP project. In addition to the design and development tasks, the programmers are responsible of making work estimations for the user stories and writing automated unit tests for everything they program. The team might also have an XP coach or a project manager who monitors the use of XP practices and keeps the work ongoing. (Cohn, 2004, pp. 233-234)

4.2.2. Scrum
Scrum is a methodology that “emphasizes a set of project management values and practices rather than those in requirements, implementation and so on” (Larman, 2003, p. 111) and it can be combined with or used to compliment other methodologies. Scrum and XP are often integrated and used together as a single comprehensive software development process. Scrum was created by Jeff Sutherland and Ken Schwaber and it was first published in 1995 (Larman, 2003, p. 135). Scrum is directed by five values: commitment, focus, openness, respect and courage (as cited in Larman, 2003, pp. 126-127). The life-cycle (see Section 4.3.) of a Scrum project is divided into 30-day iterations called sprints and each day a short meeting (the Daily Scrum) is held where the team assesses how it is doing.

There are four types of roles in a Scrum project: product owner, Scrum master, Scrum team members and everyone else. The product owner is a stakeholder representative responsible for the product and all possible feature requests are made through her. The product owner creates and maintains a product backlog, which is a list of the features that are planned to be included in the product, and also makes the decision what features will be built in each sprint (a sprint backlog is created) and reviews the system with other stakeholders at the end of the sprint. The Scrum master is a part time developer, part time leader that keeps the work ongoing and sees that the Scrum practices are followed. Team members are responsible for delivering a “potentially shippable product increment” (Sliger & Broderick, 2008, p. 295) in every sprint and updating the work breakdown structure or sprint backlog as they progress.
The first three roles are called *pigs* and are part of the actual project team doing the design, development and decisions about the product. All other people involved in the project are called *chickens*, and while they may participate in the daily meetings, only the pigs are allowed to talk. (Larman, 2003, pp. 115-119)

Cohn (2004, p. 169) lists the following main rules for Scrum:

- Each sprint must deliver working and fully tested code that demonstrates something of value to the customer.
- A sprint planning meeting is held at the start of each sprint.
- The team collectively selects the amount of work for the sprint.
- The product owner prioritizes the product backlog.
- The product backlog may be added to or re-prioritized at any time.
- Once sprint begins, only the team may add work to the sprint backlog.
- A short Scrum meeting is held every day where team members state what they did yesterday, what they will do today and what obstacles are on the way.
- Only active participants (pigs) in the sprint may speak during the daily Scrum meeting.
- The result (working software, no slide shows allowed) of a sprint is demonstrated at a sprint review meeting at the end of the sprint.
- No more than 2 hours may be spent in preparing the review meeting.

4.3. *Agile life-cycle*

Agile methodologies promote empirical processes over prescriptive processes (Larman, 2003, p. 32). Nonetheless, the iterative and incremental life-cycle of an agile project is a defined process that requires discipline to follow. The process is composed of self-contained mini projects (sprints) and releases (Sliger & Broderick, 2008, p. 323). In other words, agile projects usually run multiple sprints before the software is released, and the project can be divided into multiple releases that each have their own scope and schedule (a release per quarter for example). A release can be internal (deployed to test) or external (deployed to production). Although every sprint will not produce a release, the software should be built so that it is releasable (quality wise) after each one.

While the agile process is iterative and incremental (and not sequential), the project still transitions through different life-cycle phases (Leffingwell & Widrig, 2003, p. 28) like *inception, elaboration, construction* and *transition* (see Figure 6). Highsmith (2009, p. 82) calls these phases (respectively) *envision, speculate, explore* and *close*. 
The life-cycle phases are just names for describing the evolution of the project over time and each can contain a number of releases and sprints. The rows in Figure 6 represent different software or system development activities, whereas the columns represent sprints (and passing time). The irregular shapes or waves on each row are the relative effort used for each software development activity.

There are several ways to describe the agile development process and since different models exist, I have combined three of them into the following project life-cycle example. It goes into more detail in describing the process of actually building software in sprints and releases (see Figure 7). The process is not specific to any one agile methodology, but a generalization of what an agile process could be like.
The project starts with Sprint 0 that focuses on discovering initial requirements, project planning and setting up the project infrastructure like the development and work environment (Ambler, n.d.). Subsequent releases can have such a sprint if needed. A product backlog is created which is a prioritized and estimated list of features to be implemented in the product/project (Sliger & Broderick, 2008, p. 41). It has the same function as the requirements baseline document in traditional software projects, and it is used to manage the project scope (Leffingwell & Widrig, 2003, pp. 211-212). It is also very important to create a clear vision statement for the project to which the team can commit: it guides the development towards a common goal through the whole life-cycle of the project (Leffingwell & Widrig, 2003, p. 173).

Planning in agile projects is also iterative. At the beginning of each release, a release plan is created that is a “hypothesis about how development may proceed” (Cohn, 2004, p. 236) or “a set of goals, assumptions and decisions that guide the team in delivering value to the customer” (Sliger & Broderick, 2008, p. 40). A release plan contains those features (from the product backlog) that are going to be implemented in the release and it usually looks forward several sprints. Each sprint starts with a sprint planning session, where the work and goals for that sprint are set (sprint backlog) and the release plan is updated accordingly. The planning is done in collaboration by the whole team but it is up to the product owner to decide what is included in the sprint. A release plan can be seen as a high-level strategic plan whereas the sprint backlog is “a detailed task list that serves as a tactical guide” (Sliger & Broderick, 2008, p. 42) for accomplishing the goals of the current sprint.

Most sprints include work in all of the activities necessary for software development but the emphasis and relative effort changes over time (Larman, 2003, p. 11) as shown in Figure 6. Each sprint ends in a review meeting where working software
is demonstrated as a measure of progress and to collect feedback from the stakeholders. The team also holds a sprint retrospective where experiences of the sprint are reflected and possible actions for improving the process are agreed (Sliger & Broderick, 2008, p. 44). All releases and the project itself should also end in retrospectives of similar nature (Sliger & Broderick, 2008, p. 39).

Agile sprints are timeboxed or fixed in length even if this means dropping work out of the sprint (Cohn, 2005, p. 20). They last for a fixed duration, during which the team works on those features agreed in the sprint planning. Stakeholders can't (or in reality, should not) make changes to the sprint plan once the sprint has begun. Once the sprint has started, only the team itself can add work into the sprint if it seems there is a room for it (Cohn, 2004, p. 169). Any rework (bug fixes, changes for completed functionality etc.) is also done within the sprints. A backlog item should be created for each rework issue so that they can be included, estimated and prioritized (against new features) in sprint planning (Bayer, 2010, p. 7).

Sprint length varies and can be anything from 1-4 weeks, but shorter lengths are consider to be better. The more there are sprints in the project the more feedback the team gets from the stakeholders. This means more checkpoints where the development can be aligned to deliver the best possible business result. Teams also mature in their understanding of the methodology based on how many sprints they've undertaken rather than how many weeks they've experienced (Shore & Warden, 2008). More sprints means more exercise on the practices of the methodology. Longer sprints won't change the time the team has for the project, they just change how often progress is checked, and it takes longer to expose mistakes and recover from them. (Shore & Warden, 2008, pp. 247-248)

An agile process can also be described as a pattern of nested cycles or loops (Wells, 2001) where the contents of the project are handled on different levels of detail (see Figure 8). Elssamadisy (2008, p. 65) describes cycles in the following way:

A cycle is a period that starts with a goal and ends with an evaluation of the progress made towards achieving that goal. The evaluation results are fed into the next cycle to better achieve its goals.

Releases and sprints are obviously the major cycles in an agile project, but daily work also follows the same pattern and includes planning and feedback loops. For example, each day a short stand-up meeting (or a Daily Scrum) is held where the team coordinates its work, synchronizes daily efforts and assesses and revises its plans (Cohn, 2005, p. 20)
Software development projects (whether agile or not) do not exist in a vacuum although the processes and the project life-cycles are often described without the surrounding systems development or the organizational context. Agile methodologies concentrate on software projects and software development and often have little to say about portfolio or product management or the maintenance and retirement of a software system (Ambler, n.d.). These topics and processes are out of the scope of this thesis, but they still need to be considered, planned, operated and managed in a software development organization.

4.4. Agile practices and methods

All agile software development methodologies include a number of different practices and techniques that define how requirements, design, development, testing and project management should be done. Many of the practices are shared between the different methodologies.

There are more practices (for example, see Appelo (2009) for a comprehensive list) than could be introduced here so I'll summarize some of the more significant ones. These have been collected from various sources (Cohn, 2004; Elssamadisy, 2008;
Larman, 2003; Shore & Warden, 2008) and are presented in Table 4. The table is divided into sections according to the five activities performed in software development projects. Although some practices can be hard to place in any single category, I've done so for the sake of clarity. Each section has a number of agile practices that are all grouped in the column named Practice. Each has a short description and any specific work product they produce including written documentation is mentioned.

Table 4: Agile practices and methods.

<table>
<thead>
<tr>
<th>Practice</th>
<th>Description</th>
<th>Work products</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small releases</td>
<td>The development progresses in a series of short, timeboxed sprints, each producing new fully functional features and working software. Work cannot be added to a sprint once it has started.</td>
<td>Backlogs: Product, Release, Sprint Burndown chart Story Cards</td>
</tr>
<tr>
<td>Planning</td>
<td>Planning is done on several levels in an agile project (see Section 4.3) and it usually means the process of creating, choosing and elaborating the next work items, such as user stories for example, for the next cycle (project, release, sprint, day). Planning usually involves a backlog which is a documented list of prioritized work items and their work estimates. Planning poker is a specific sprint planning technique in which the whole team participates.</td>
<td></td>
</tr>
<tr>
<td>Sustainable pace</td>
<td>Work should be done at such a pace that can be sustained indefinitely. People create better, higher quality products when they are healthy, motivated and enjoying their work. Frequent overtime is discouraged as it causes opposite effects and is seen a sign of deeper problems.</td>
<td></td>
</tr>
<tr>
<td>Self organizing teams</td>
<td>During the sprint, the team itself is responsible for meeting the sprint's goals and has the authority to plan and execute its work as it sees fit. The role of the project manager is not to direct the team but to help it achieve its goals.</td>
<td></td>
</tr>
<tr>
<td>Retrospective</td>
<td>Retrospectives are meetings held after sprints, releases and projects where the project team reflects its experiences and decides on possible actions for improving the process.</td>
<td></td>
</tr>
<tr>
<td>Daily meeting</td>
<td>Each day a short meeting is held where the team coordinates its work, synchronizes daily efforts and assesses and revises its plans. The Daily Scrum (or stand-up) is specific technique related to Scrum.</td>
<td></td>
</tr>
<tr>
<td><strong>Research</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-site customer</td>
<td>The whole team works together in a common project room with an on-site customer or a customer representative like a product manager. The on-site customer is a subject matter expert working on the requirements with the developers and is empowered to make decisions about the requirements. Having an on-site customer enhances communication helping to develop better quality requirements and reducing documentation overhead.</td>
<td>Story cards</td>
</tr>
<tr>
<td>User stories</td>
<td>User stories (or just stories) are a requirements engineering tool in agile projects. They are short written descriptions of features used for planning and as reminder, conversations to flesh out the details of the feature and tests that document and determine that the feature is complete. They also include a work estimation. User stories can be documented on paper index cards or in an appropriate software tool. In many ways user stories replace the traditional requirements documents.</td>
<td>Story cards</td>
</tr>
</tbody>
</table>
## Design

| Simple design | Simple design means that the team aims to have the simplest possible design that is enough to deliver the features the customer needs. Simple doesn't necessarily mean simplistic but rather that the system should not contain anything unnecessary for its intended goal. |
| Metaphor | Metaphor supports the idea of having simple design by providing a frame of reference for how the team should think about the system and thus aiding communication about the design. |

## Programming

| Pair programming | Pair programming is a practice where two developers program together (actually sharing a computer) in order to produce better quality code and to share an understanding of the code. One developer writes the code thinking a few lines ahead, while the other one watches, thinks about future solutions and helps to solve problems. The developers switch roles often, e.g., a few times in an hour and also pairs are switched on a daily basis. |
| Refactoring | The program code should be continuously cleaned and restructured to achieve a better and simpler design without changing the external behaviour / functionality of the system. |
| Team code ownership | All code is owned by everyone and any pair of programmers can change any code. This collective responsibility encourages problems to be fixed as they are spotted and development is faster as the "bottleneck" of individual code ownership is removed. |
| Coding standards | The team agrees and follows a set of common standards on how to write program code which improves maintainability and readability of the code. |

## Testing

| Test-driven development (TDD) | In test-driven development a unit test is written before the actual program code that passes the test. Testing can be automated by writing test code that runs the actual operational code. Developers follow a short cycle where they actually test first and if the test fails only then they write the operational code to make the test pass. This approach helps to create a comprehensive set of tests for the system and ensures good code quality. |
| Continuous integration | All code checked in to a version control system is automatically and continuously re-integrated and tested on a separate build machine. Doing integration testing continuously and automatically lets the team find related problems when they are created (and not late in the project) and reduces the amount of repetitive manual work needed to integrate the software. |
| Customer acceptance testing | Customer representatives write acceptance criteria for the user stories they create which then can be automated by the developers. These tests are then run to see if the related feature was developed as defined in the user stories. |
| Sprint review | At the end of each sprint, a meeting is held where the results of the sprint are demonstrated to stakeholders in the form of working software. The goal is to share information, evaluate the design and functionality of the software, get feedback and brainstorm future directions. |

The usage of these and other agile practices has been studied with surveys. VersionOne, a company developing agile project management tools, run a survey (VersionOne survey, 2008) with 2031 respondents. One of the questions of the survey asked which practices the respondents employ in their agile methods. Based on the answers the most commonly used practices are listed in Table 5.
Table 5: Usage of agile practices (VersionOne survey, 2008).

<table>
<thead>
<tr>
<th>Agile practice</th>
<th>% of respondents using the practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint planning</td>
<td>86%</td>
</tr>
<tr>
<td>Unit testing</td>
<td>77%</td>
</tr>
<tr>
<td>Daily standup</td>
<td>75%</td>
</tr>
<tr>
<td>Release planning</td>
<td>72%</td>
</tr>
<tr>
<td>Continuous integration</td>
<td>65%</td>
</tr>
<tr>
<td>Retrospectives</td>
<td>59%</td>
</tr>
<tr>
<td>Refactoring</td>
<td>59%</td>
</tr>
<tr>
<td>Coding standards</td>
<td>57%</td>
</tr>
<tr>
<td>Test-driven development</td>
<td>49%</td>
</tr>
<tr>
<td>Collective code ownership</td>
<td>41%</td>
</tr>
<tr>
<td>On-site customer</td>
<td>34%</td>
</tr>
<tr>
<td>Pair programming</td>
<td>31%</td>
</tr>
</tbody>
</table>

Ambler (2009) has also conducted a survey about the usage of agile practices. The survey had 123 respondents. The respondents had to choose seven agile practices (out of the total of 30) they had found out to be the most effective. According to the results the following eight practices (see Table 6) were believed to be the most effective.

Table 6: Most effective agile practices (Ambler, 2009).

<table>
<thead>
<tr>
<th>Agile practice</th>
<th>% of answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous integration</td>
<td>65%</td>
</tr>
<tr>
<td>Daily stand up meeting</td>
<td>47%</td>
</tr>
<tr>
<td>Test-driven development</td>
<td>47%</td>
</tr>
<tr>
<td>Sprint planning</td>
<td>44%</td>
</tr>
<tr>
<td>Refactoring</td>
<td>43%</td>
</tr>
<tr>
<td>Retrospectives</td>
<td>39%</td>
</tr>
<tr>
<td>Pair programming</td>
<td>36%</td>
</tr>
<tr>
<td>Active stakeholder participation / on-site customer</td>
<td>35%</td>
</tr>
</tbody>
</table>

Scrum and XP are often used together as their practices support each other well and alone the methodologies might not be comprehensive enough for an organization's software development process needs. Each methodology was created for a specific reason. XP practices concentrate on the technical aspects of software development where Scrum practices concentrate on organizing the overall structure of the project. (Aguanno, 2004, p. 288)

Although individual Scrum practices can be used to support XP, it is recommended (Cohn, 2004, p. 234) to adopt all the XP practices as such. XP practices work in synergy enabling each other and “it is risky to customize XP by removing some elements” (Larman, 2003, p. 141). Wells (2009c) writes that “Extreme Programming is
a lot like a jigsaw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen.”. Only after the team becomes experienced in XP it should start thinking about modifying the practices (Cohn, 2004, p. 241).

Each organization has different needs and requirements for their software development process that relate to business and product development models, technology and people. One of the four values of the Agile Manifesto is “Individuals and interactions over processes and tools”. Thus people should not be enforced to follow a common set of processes and practices but the processes and practices should be molded to the team itself (Highsmith, 2009, p. 125). Rather the organization should provide a framework for the teams to organize their work. Within those boundaries the team can then choose which practices it will use depending on their and the project's needs. The team should wait to modify practices until they've tried them for a couple of sprints (Highsmith, 2009, p. 125). This of course requires experience in the practices and understanding how the changes will affect the overall development process. In the end the agile practices are only a manifestation of the agile philosophy and Highsmith writes that “Agility is principally about mindset, not practices.” (2009, p. 18) and that “A team can employ Agile practices, but it won't achieve the potential benefit of Agile development without embracing Agile values and principles.” (2009, p. 26).

4.5. Agile benefits and criticism

Agile methodologies were developed as a remedy to the failure of predictable manufacturing concepts, such as the waterfall life-cycle, big up-front specifications and speculative planning as they were misapplied to software development. Besides giving flexibility and focusing on delivering customer value, agile methodologies reduce the risk of building a wrong product by:

- working on the requirements with an on-site customer,
- eliciting stakeholder feedback early and often with working software, and
- adapting development to changing requirements based on that feedback.

Agile development also reduces the risk of building the right product wrong with test-driven development, continuous integration and other practices and techniques concentrating on software quality. When working software is evaluated and tested in every sprint, requirements and design issues and also software defects are discovered much earlier than in waterfall type projects where testing is done only once at the end of the project. Also the risk of getting stuck in the requirements or design phase in an unclear project is negated as agile development ensures that actual implementation is done in every sprint. (Aguanno, 2004, pp. 25-34)

Agile software development also improves the project manager's and the client's control over the project in situations where requirements change a lot. Project
communications are enhanced with co-located teams, on-site customers and daily meetings making the project status more transparent and allowing fast decision making. Frequent delivery of working software makes progress objectively measurable and gives the stakeholders a chance to provide feedback early and thus redirect the project priorities accordingly. Early feedback lowers the cost of changes made to the product. The software product is also potentially releasable after every sprint which means that the time to market is decreased. Potential issues in the business are surfaced earlier and in the case the project is doomed to fail, monetary losses can be minimized by terminating the project early. (Aguanno, 2004, pp. 35-40)

Larman (2003, pp. 51-54) lists the following benefits that are related to agile development and iterative development in general:

- Iterative life-cycle is associated with lower risk and better success, productivity and defect rates.
- Risks are mitigated and discovered early as risk-driven iterative development forces tackling the hardest features, issues and problems first.
- Iterative development accommodates and provokes early change which is suitable for software product development.
- Team and customer confidence and individual satisfaction grow as visible progress is demonstrated in every sprint, leading to a sense of completion, competency and closure.
- Early working demos and public releases of the software are possible, which creates new business opportunities.
- Progress tracking and predictability are more reliable as they are based on similar completed sprints and more importantly, on tested working software.
- Early testing gives higher quality and less defects.
- Early evaluation and feedback from clients and users ensures that the final products match their actual needs.
- Team's self-assessment in every sprint provides early and regular process improvement.
- Team communication and coordination and client engagement is forced early on which improves the chances for the project being successful.
- Short timeboxed sprints increase productivity and force the team to work on a feature subset of the software that is manageable in complexity. Prioritization of the features is also forced early on as stakeholders need to decide what will be included in each sprint.

According to the VersionOne survey (VersionOne survey, 2008) the 2031 respondents have benefited in different ways from implementing agile practices (see Table 7). The percentages on the right column refer to those respondents who felt that
Agile practices have “improved” or “significantly improved” the item on the left column.

Table 7: Realization of agile benefits (VersionOne survey, 2008).

<table>
<thead>
<tr>
<th>Benefit gained from agile practices</th>
<th>% of respondents saying “improved” or “significantly improved”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved project visibility</td>
<td>83%</td>
</tr>
<tr>
<td>Improved team morale</td>
<td>74%</td>
</tr>
<tr>
<td>Increased productivity</td>
<td>74%</td>
</tr>
<tr>
<td>Simplified development process</td>
<td>69%</td>
</tr>
<tr>
<td>Enhanced software quality</td>
<td>68%</td>
</tr>
<tr>
<td>Enhanced software maintainability / extensibility</td>
<td>66%</td>
</tr>
<tr>
<td>Improved alignment between IT and business</td>
<td>66%</td>
</tr>
<tr>
<td>Accelerated time-to-market</td>
<td>65%</td>
</tr>
<tr>
<td>Reduced risk</td>
<td>65%</td>
</tr>
<tr>
<td>Enhanced ability to manage changing priorities</td>
<td>63%</td>
</tr>
<tr>
<td>Improved / increased engineering discipline</td>
<td>60%</td>
</tr>
<tr>
<td>Reduced cost</td>
<td>38%</td>
</tr>
<tr>
<td>Management of distributed teams</td>
<td>29%</td>
</tr>
</tbody>
</table>

Agile Adoption Rate Survey (Ambler, 2008) also suggests similar benefits from adopting agile development. From the 350 respondents (to the corresponding questions) a majority felt that agile development was more effective than traditional approaches:
  - 82% thought that productivity was “somewhat higher” or “much higher”
  - 77% thought that quality was “somewhat higher” or “much higher”
  - 88% thought that business stakeholder satisfaction was “somewhat higher” or “much higher”.

Despite its possible benefits, organizations and people may have concerns about agile development before they start to practice it. Some are valid (Keefer, 2004) but some are common myths that have been used to criticize agile development: the lack of up-front planning, documentation, predictability, and engineering discipline, the loss of management control and the inability to scale (VersionOne survey, 2008). It is true that methodologies like Scrum and XP did not have all the answers to software development when first introduced (and still don't (Taipale & Tanninen, 2009)). Since those days, though, the agile community has learned to produce answers to these questions. Probably hundreds of books have been written about agile development so it is not a challenge to find information on aspects of software development that the specific methodology descriptions do not deal with. Agile culture encourages teams and organizations to develop practices and specific ways of working that are needed. If a methodology does not describe something the team needs in their development process,
the team can then add necessary practices to the methodology and improve their process.

Agile methodologies have also been criticized that the practices and ideas they are proposing are not new. This is true: for example iterative and incremental development has been used since the 1960s. But what is new in agile methodologies is that they view these practices and principles as a whole and this has proven to be a success for those techniques that were previously considered hard to understand. (Larman, 2003, p. 35)

Aguanno (2004) also points out two issues related to agile development that need to be considered. Firstly, a self organizing, empowered agile team tends to locally optimize their way of working in a particular project, which can cause problems in enterprise project/portfolio management. Secondly, agile methodologies are not formal enough for life-critical systems development as they lack the necessary design reviews and evaluations needed to discover possible safety issues. (Aguanno, 2004, p. 296)

There is a lot to be gained by adopting agile methodologies, but organizations and people need to realize it is not a magic cure or a silver bullet. 76% of the 2031 respondents of the VersionOne survey (VersionOne survey, 2008) said that 75% or more of their organization's agile projects are successful. Although the number is far more encouraging than the 16% success rate reported by the initial CHAOS report (The CHAOS report, 1995) it is a good reminder that agile methodologies (and people) are far from perfect when put to organizational context. The survey identified the following issues as the leading causes of project failure:

• company philosophy or culture conflicting with core agile values,
• lack of experience with agile methods,
• external pressure to follow traditional waterfall phases and practices,
• unwillingness of team to follow agile practices,
• lack of cultural transition,
• lack of management support, and
• insufficient training.

Changing the way people and organizations work is never easy. Agile adoption needs a clear and well stated adoption program that tackles these issues and makes the change possible (see Section 7.2. for more about adoption).
5. Similarities and differences in UCD and agile development

5.1. The challenge of agile UCD

Since the Agile Manifesto was written in 2001, agile methods have become mainstream in the software development industry. This can be seen as a progress for usability as agile development can overcome the problems of traditional methods that impeded usability in the past (Nodder & Nielsen, 2009, p. 7). Despite this there are challenges in integrating UCD and agile development. Agile methodologies started as methods for programming and because of that teams are often uncertain how non-coders fit into the agile development team (Bayer, 2010, p. 9). Nodder and Nielsen write in their report that “A narrow approach to Agile as a programming methodology instead of a system development methodology does threaten to destroy the last decade's progress in integrating usability and development.” (Nodder & Nielsen, 2009, p. 7). Of course, change resistance from people and organizations is something that affects the integration whether it is to be done with agile or more traditional software development methodologies.

Both UCD and agile development have the same goal of producing high quality software and evidently they can. They share some common aspects, but also have differences in philosophy and practice. Blomkvist (2005) has studied agile software development from the UCD perspective and writes that it has “qualities that can provide a solid foundation for user-centered attitude: focus on people, communication, customer collaboration, adaptive processes and customer/user needs” (Blomkvist, 2005). Regardless of these qualities, agile development cannot be considered to be user-centered as its values do not have the necessary focus on users and usability. Some aspects of agile development could be even seen infertile for UCD:

Furthermore, some of the agile processes' prioritized areas of interest can prevent a user-centered attitude: a focus on programming and programmers, automated tests, very short iterations and fast increments, and executable software as a measure. Other problem areas are the confusion between users and customers, unsatisfactory techniques for modelling users and tasks (i.e., user stories and use cases), the fear of early design, as well as insufficient activities for interaction design. (Blomkvist, 2005)

Although usable software can be developed with agile methodologies, it can be just a coincidence, as neither XP (Jokela & Abrahamsson, 2004) nor Scrum (Singh, 2008) adequately address usability needs. They lack the practices to systematically do so (see Section 4.4.). The following sections explore the similarities and differences of
agile development and UCD and how practitioners in the industry have overcome issues between the two.

5.2. **Iterative process**

Often UCD tends to be iterative but not incremental as the whole system is designed completely and then refined in later iterations. Delivery (handover to development) is made once. Agile development, on the other hand, is incremental as the system is developed and released in small pieces. In a perfect world agile development would not be iterative as re-iterating already completed functionality means rework (and wasted resources) and is against the agile idea of 'maximize the work not done'. Agile development in the real world is iterative as it is rare for a piece of software to be 100% right the first time it's considered 'done'.

While both are iterative, there is still a difference in what is being iterated. In UCD iterations, larger systems are designed and evaluated at different levels of fidelity before implementation but in agile development the focus is strictly in building and releasing working pieces of the larger system (Blomkvist, 2005). In this light UCD has to change and become incremental, since there might be no time to do holistic up-front design within the agile process.

The goal of an agile sprint is to deliver a new piece of working functionality and with such a quality that the software could be released to users if so decided. Postponing testing, bug fixing and refactoring creates technical debt that eventually has to be paid before the release. Similarly at the end of each sprint, the UI should be operational, bug free and usable. Failure to do so prevents the software from being delivered, thus the whole team has a stake in ensuring that the necessary UCD work gets done. (Bayer, 2010, p. 13)

5.3. **User vs. customer as a stakeholder**

One of the biggest differences of agile and UCD is the notion of the user as a stakeholder and how software requirements are developed. In UCD users are people that actually use the software and that participate in requirements gathering through interviews and field studies (user research) and in the evaluation of the design and implementation through usability tests. Agile development also values collaboration with users and considers stakeholder participation in the development process as the most effective way of communicating needs to the developers. Stakeholders should be co-located (the on-site customer) with the developers in order to take part in defining and testing the software on different levels. Frequent delivery of software increments also makes it possible for the stakeholders to evaluate the software under real conditions. (Blomkvist, 2005)
According to Blomkvist (2005) these “practices sound promising from a UCD standpoint” but “a common stumbling block is that the agile processes seldom distinguish between customers and users – all too often, they are regarded one and the same” (Blomkvist, 2005). A stakeholder might be an actual end-user for the software being developed but often she is a domain-expert or a product owner representing the customer organization and might not use the software at all. Agile development focuses on satisfying the customer's needs and changing business requirements but leads to problems if there is no one on the team that can truly represent the end-user (Nodder & Nielsen, 2009, p. 35). Close co-operation between developers and the customer does not ensure good usability even if the finished software can be considered to fill quality requirements otherwise (Jokela & Abrahamsson, 2004).

Even if the on-site customer is an end-user representative, agile methodologies still lack the necessary practices for designing usability. The on-site customer is supposed to define requirements by writing and prioritizing user stories and specifying acceptance tests but there are problems in this approach. Firstly, user stories as format fail to capture many aspects of the actual user interaction (Blomkvist, 2005). Although their relation to the context of use could be improved by writing top level scenarios (Obendorf & Finck, 2008), additional design tools are needed (see Section 5.5.). Secondly, users “can express what they need to a certain extent, but on their own, it is difficult for them to actually design a new system” (Blomkvist, 2005). What users think they want and what they actually need are two different things and that is why one of the primary UCD guidelines is to “watch what users do, not listen to what they say” (Nodder & Nielsen, 2009, p. 5). There is also a risk that the developers will transfer the responsibility of the system’s usability to the on-site customer (Blomkvist, 2005). Software projects are often so long that the end-user representatives participating in daily development can become too exposed to the inner workings of the software and do not truly represent the actual users anymore.

A solution to these problems could be to hire a UCD specialist to act as a communication link between the business stakeholders, users and the developers, and help, e.g., by creating personas. For example Extreme Personas (Wolkerstorfer et al., 2008), that are like traditional personas but created incrementally and iterated through the project, could help the team to focus on the end-users. These personas could also be used to prioritize features and user stories in backlogs during release and sprint planning.

Also a design/development partner program could be created for recruiting end-user representatives that commit in helping the team through the whole project (Sy, 2007; Williams & Ferguson, 2007). This way end-user representatives would always be readily available for user and usability studies.
5.4. Generalist vs. specialist: Role of the UCD specialist in a team

Agile development emphasizes effective teamwork, project management and organization culture which relate to the UCD principles (see Table 1) of professional attitude, usability champion and process customization. Thus the following values are also considered important in UCD (Blomkvist, 2005):

- Communication between people is essential, and face-to-face conversations are preferred. Cooperation and sharing of responsibilities between business people and developers is vital.
- Build projects around motivated and skilled individuals; trust them and give them the support and the environment they need.
- Organize the project into small, effective, multi-disciplinary teams, where collaboration and communications are present on all levels. Self-organizing teams yield the best results.
- Promote sustainable development. The people involved should be able to maintain a constant pace without burning out.
- The team should reflect on how to become more effective at regular intervals, and fine-tune its behavior, if needed.

These guidelines result in empowering the various roles in a project to make decisions and take responsibility for their area of profession. The projects should have skilled UCD specialists with the authority to rule on matters affecting usability during the system's life cycle. In practice, many agile processes have a narrow view on what competencies are needed in a project, and UCD specialists are routinely overlooked. Without them the chances of producing usable systems are slight. This is a serious weakness affecting most out-of-the-box agile methodologies. If they were aware of usability as a concept, the agile values could be used to promote the idea that teams should be empowered, multidisciplinary (and gross-functional) and include UCD specialists. (Blomkvist, 2005)

Ideally, each project would have a dedicated and co-located UCD specialist working as full team member (a pig in Scrum terms). In reality the UCD specialist is often working on multiple projects at the same time and needs to prioritize time between them based on what is going on in each at the moment. (Bayer, 2010, p. 24)

These UCD specialists should own user experience issues, i.e., make the related design decisions and share the information with the developers and guide them to the right path. The UCD specialist acting as link between the product owner, the users and the developers not only ensures that business goals and usability are being addressed accordingly, but also helps to change the image of the UCD specialist to a valuable resource (from someone who just points out all the wrongs). To enjoy the benefits of agility and close co-operation, the UCD specialists should sit together with the rest of
the project team. Co-location also supports the idea of cross-functional teams and increases the general awareness of usability and UCD and their perceived value in the organization as teams are exposed to them in practice. (Nodder & Nielsen, 2009, pp. 47-48, 108)

Singh (2008) suggests that the UCD specialist should not be a regular member of the team, but act as a second product owner instead. While the first (traditional) product owner focuses on the features of the product, its marketing and sales, the other one focuses on usability and user experience. The two should work as peers and make decisions together. This idea is based on the argument that traditional product owners neither have time nor the experience to address usability concerns. There is also a concern that as a regular team member the UCD specialist might not have enough influence or leverage to give usability high enough priority. The UCD product owner is responsible for the user experience vision of the product which is realized through personas, user goals, navigation models etc. (Singh, 2008)

Singh's suggestions support the practice of up-front user research (see Section 5.5.) and that the product vision is properly defined before the implementation begins. However, there is a concern that the team might not see the benefits of UCD work or adopt a user-centered attitude easily if usability is 'mandated from above'. The UCD product owner still needs to work as close as a team member would together with the developers to ensure good usability and to evangelize User-Centered Design. Chamberlain, Sharp, & Maiden (2006) have witnessed power struggles between the different disciplines and suggest that some kind of a balancing mechanism might be needed to ensure that designers and developers have equal power within the team. All members of the team should also be involved at each key point of the project (cross-functional team).

While agile teams need to recognize the importance of dedicated UCD specialist as a role, the UCD specialist must also adopt the agile mindset and adapt to the development process to work efficiently. The fast, iterative and feedback oriented working environment of agile methodologies might be something that the UCD specialists have not been used to and forces them to change their working practices. Rather than doing all their work before the development begins, UCD specialists must build their designs incrementally and refine them concurrently as the actual software is being built (Ramsay, 2010). This can feel strange for those with more traditional backgrounds, but design in agile development should be a collaborative effort where the UCD specialist becomes the design facilitator (Patton, 2008).

5.5. Up-front design
One of the principles of agile development is to 'maximize the work not done'. Agile development favors light-weight just-in-time modeling instead of extensive up-front
design, which could turn out to be a wasted effort when requirements change. A question is whether this view also applies to up-front UCD work as agile methodologies only discuss technical design.

While agile methodologies are considered effective in projects with unclear requirements, they have actually little to say about how those requirements should be gathered and made clear in the first place. Agile methodologies do not cover how to do systems or requirements analysis (Bayer, Holtzblatt, & Baker, 2004). Agile teams can cope with the symptoms of unclear requirements, but do not have the means to fix the larger cause (Colfelt, 2010). The team expects the customer to deliver the requirements, but if that does not work out, the team just has to build something and ask if the outcome was what the customer expected. As changes to requirements are more expensive after the implementation than before it, not doing at least some up-front requirements analysis would seem to contradict the idea of maximizing the work not done.

UCD has traditionally been done up-front as there was little chance to make changes after development had started (Nodder & Nielsen, 2009, p. 34). Although agile development is against big design up-front there are other problems (besides cost) in not doing at least some UCD work before the implementation. The user interface can seem piecemeal as it is built feature by feature without a holistic view of the software being developed. Thus high-level UCD work (like user research, requirements gathering and concept design) is required to overcome this risk and it involves working with the project management team including sponsors, stakeholders and business representatives. When development begins, the focus of UCD turns to user-interface-specific work. A balance must be struck between “knowing enough high level UI guidance and going too far into the per-screen design process” (Nodder & Nielsen, 2009, p. 49). High level UI modeling tasks might include top-level storyboards, wireframes for key interactions, design patterns for reusable UI widgets and a style guide for graphical design. Detailed up-front design on user story level makes little sense before it's known what will be developed, which might not be clear until the sprint begins. If it is not possible to do up-front UI design before development, the development should start from back-end or non UI-intensive features buying time for the UCD work. (Nodder & Nielsen, 2009, pp. 34, 46, 49, 52)

According to Ferreira, Noble, & Biddle (2007) the XP teams that do interaction design understand the work to include a lot of business and end-user analysis. Up-front UCD work is needed for gathering and validating requirements together with the stakeholders. It is considered to save a lot of time and resources instead of being waste. Prototyping possible solutions with ‘pen and paper’ is seen more cost-effective than making the same analysis with working software. (Ferreira et al., 2007)
Similarly to up-front design work, documentation in agile development is done only when considered absolutely necessary, i.e., the effort spent pays back by reducing future (unnecessary) work. Interactions and interfaces remain speculative until they've been implemented and describing them in documentation might be waste. Face-to-face communication about the UI is seen more effective than documentation during development. The on-site customer might be able to define business requirements and logic for the software this way, but it still leaves a lot of guessing what will be actually built, how it should work and what it will look like. The UCD specialist can provide answers to these questions not only by communicating user needs, but by assisting the communication with storyboards and wireframes. In the absence of traditional software specification documentation these documents will become the key way of communicating design concepts to the team. These tools effectively become the specification as they document the design decisions that have been made. They also help the stakeholders to understand how the software will look like before it is implemented, and with little effort they can be used in usability testing as well. This pre-development evaluation (see Section 5.6.) saves work as developer time is not wasted in building the thing wrong. (Nodder & Nielsen, 2009, pp. 52-54)

5.6. Testing
In agile development there are two different types of testing that are done to ensure the quality of the software. Firstly, there is automated developer testing (on unit and integration levels) which is a prerequisite for producing high quality code in agile processes (Blomkvist, 2005). Automated testing is done on a daily basis. Secondly, there is customer acceptance testing where the customer/user representatives test the actual working software to see if it fulfills the requirements. This can be done at the end of each sprint or as soon as a reasonable set of functionality like a user story is complete. Agile development enables testing of real software early and throughout the project, but for the software to be usable, the agile testing practices need to be complemented with usability evaluations (Blomkvist, 2005).

It seems that the production of working software at quick and constant pace provides a great setting for usability evaluations, but the traditional usability testing conducted in a laboratory hardly fits this process. While there is little time to do usability testing, there are many discount methods that can be used in agile development (Nodder & Nielsen, 2009, p. 36). For example, usability criteria could be included in user stories and in separate usability stories which would then be a part of customer acceptance testing (Carbon, Dörr, & Trapp, 2004). The developers can also do basic UI evaluation with the aid of style guides or heuristics and cognitive walkthroughs.

Although developer or expert evaluations will enhance the usability of the software, it also needs to be tested with real end-users that have the understanding about
the domain. Usability testing should be done throughout the project and ideally at least once before development begins. This up-front testing could be done with paper prototypes or wireframes for example. Recruiting real end-user representatives can be an effort and in those cases up-front tests could be conducted with colleagues, 'saving' the actual end-user test participants to tests done with the actual software (Sy, 2007). A design partner program could be effective to overcome the recruiting issues.

Preparing a usability test can involve a lot of work, and if the decision of whether to test or not has to be made each time, there is a risk that testing won't be done. The project team might not see the benefits of testing. The preparation effort can be decreased by outsourcing the recruitment of test participants and by making the testing a recurring event, e.g., once every other sprint. It may seem strange to schedule testing before knowing what to test, but on the other hand there should always be some new features to test in the agile development process. This revolving door testing helps the team to decide to do testing and allows it to focus on the content of study (Nodder & Nielsen, 2009, p. 80). For example, Steve Krug (2010, p. 23) suggests an approach where usability testing is scheduled to be done once a month with three users and the development team is debriefed immediately after on the same day.

Besides schedule issues, the way the findings of the usability study are reported to the development team also needs to be reconsidered in agile development. Traditionally a lengthy test report, with all the issues and suggestions how to fix them, would be written after the test. This is not an effective strategy in agile development as it takes more time to write the report than the already busy UCD specialist has. There is also the chance that the developers won't bother reading a lengthy document, and a risk that some of the features the report discusses have already been changed in the mean time. The effort spent on usability testing and writing the report is wasted, if the issues found are not fixed.

A more proper approach for agile development would be solution focused usability testing. The team could observe the study through a screen sharing application and discovered usability issues would be written down and discussed face-to-face right after the study (Krug, 2010, p. 103). Those issues that can be fixed, are fixed on the spot and the rest are reported with issue cards (Sy, 2007) as rework items and fixed in later sprints depending on the backlog prioritization. One way of doing usability testing that focuses on solving usability issues, is the Rapid Iterative Testing and Evaluation or RITE methodology (Medlock, Wixon, McGee, & Welsh, 2005) where usability fixes are done between test sessions.
6. Integrating UCD into agile development

6.1. Integration principles

Although agile development can't be considered to be user-centered, it should be possible to integrate its basic values and principles with UCD which in turn can benefit by becoming more agile. The challenge is to keep the methodology agile and shape the UCD practices to be light-weight as well (Jokela & Abrahamsson, 2004). Many have succeeded in the integration, Usage-Centered Design (Constantine, 2002) and Rapid Contextual Design (Bayer et al., 2004) as examples. There is no reason “why Agile processes could not be customized or adapted to UCD, or vice-versa” (Blomkvist, 2005) but “the diversity of the various Agile methods and techniques makes it impossible to make any blanket recommendations” (Blomkvist, 2005). Each agile methodology and its practices should be examined to determine if they can be integrated with UCD practices and what the consequences will be. After the initial examination there are three different approaches that can be taken for the integration (Blomkvist, 2005):

1. UCD practices are added in to agile development framework. This allows organizations, that are already doing agile development, to deliver more usable systems without abandoning their established methods. The disadvantage is that the role of the UCD might not get comprehensive enough to deliver value to its full potential, if just a few separate practices are taken into use.

2. Agile practices are added in to the UCD framework. This approach aims to make the UCD process more agile, not necessarily to create a comprehensive software development framework.

3. Balanced integration between agile development and UCD. Agile development and UCD are combined in a balanced way. This happens by creating a new hybrid software development methodology from ground up or just welding the two approaches together. The values, principles and practices of the both are integrated.

The question which approach is best depends both on the development and user organizations, the processes currently in use and the people involved. Although the balanced integration might be the hardest one of the three to achieve, it is required for maintaining the core values of both agile and UCD. After that the coordination of practices, people and process is more likely to succeed (Blomkvist, 2005). Blomkvist's suggestions on how to achieve a balanced integration are summarized in Table 8.
Table 8: Suggestions for achieving a balanced integration between agile development and UCD (Blomkvist, 2005).

<table>
<thead>
<tr>
<th>#</th>
<th>Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The prime objective is delivering working and usable software. Usability activities are important, but there must be a balance between releasing products and improving usability. Releasing in increments is beneficial because the developers can learn from actual usage.</td>
</tr>
<tr>
<td>2</td>
<td>Users (assisted by the usability designers) should be in charge of the prioritization the system's features to achieve high degree of usability.</td>
</tr>
<tr>
<td>3</td>
<td>Both developers and users involved in the development process are important and how the team and work practices are organized must be taken into consideration. Ultimately, people are more important than processes.</td>
</tr>
<tr>
<td>4</td>
<td>Actively involve users (not just customers) in all phases of development by co-locating them with the developers and/or by studying the details of users' context by visiting their workplace.</td>
</tr>
<tr>
<td>5</td>
<td>Development projects require skilled usability designers who should be empowered to make decisions about matters that affect the system’s usability.</td>
</tr>
<tr>
<td>6</td>
<td>Improve team communication by working in pairs. Pairing users with usability designers and/or usability designers with programmers will contribute to boosting communication.</td>
</tr>
<tr>
<td>7</td>
<td>Usability validation is needed at different stages of the development cycle. If users are ready at hand, simple low-scale usability tests can be performed early and frequently by using lo-fi prototypes. These basic usability tests must nonetheless be complemented with more thorough usability acceptance tests in which more complete parts of the system are validated.</td>
</tr>
<tr>
<td>8</td>
<td>Evolutionary development is essential as complex or tacit aspects of the system are difficult to specify in advance. The process must allow changes in the requirements.</td>
</tr>
<tr>
<td>9</td>
<td>The process should include suitable methods to determine and specify usability goals that can direct the course of the iterative development. Usability designers are required to complete a full analysis together with users and transform the information into usability goals and design criteria.</td>
</tr>
<tr>
<td>10</td>
<td>An evolutionary approach must be combined with early and coordinated usability design activities, such as user and task analysis, personas, scenarios, conceptual models and paper prototypes and iterative design.</td>
</tr>
<tr>
<td>11</td>
<td>Relatively simple models such as paper prototypes and lo-fi mock-ups should be used as part of development to create and test design solutions in rapid iterations. It is more effective to develop and test simple prototypes than released increments of the system. The added value of simple prototypes is the possibility to quickly explore and evaluate different design solutions.</td>
</tr>
</tbody>
</table>

Nodder & Nielsen (2009) favor an integration approach which might be more practical in most organizations. They write that “The trick as a user experience practitioner is to fit within whatever process the team is following, and to suggest changes which can show demonstrable improvements to the user experience.” (Nodder & Nielsen, 2009, p. 22). Constantine (2010a) has also witnessed this approach in his work and calls for a more balanced integration where the two disciplines would be on more equal terms. He argues that agile development practices and teams also need to change to make the relationship work better and that in some cases it might be even better, if UCD would drive the development process (Constantine, 2010b). Knowledge sharing between the disciplines is crucial for developing mutual respect and is required for building better products.

By integrating UCD and agile development, a systems development process is born which offers several benefits over any particular agile methodology or traditional approaches where UCD is integrated into a waterfall process. For example, timeboxed
sprints require the team to focus on the most important aspects of the software and offer a good setting for conducting usability investigations throughout the whole project. Several other benefits, reported by Miller (2005), Nodder & Nielsen (2009, p. 21) and Sy (2007), are collected in Table 9.

Table 9: Benefits of integrating agile development and UCD. (Miller, 2005; Nodder & Nielsen, 2009, p. 21; Sy, 2007)

<table>
<thead>
<tr>
<th>#</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Iteration of requirements is possible (as opposed to waterfall).</td>
</tr>
<tr>
<td>2</td>
<td>Contextual inquiries and usability testing can be conducted during the same site visit.</td>
</tr>
<tr>
<td>3</td>
<td>User stories and the backlog can be enhanced and prioritized according to user research.</td>
</tr>
<tr>
<td>4</td>
<td>Larger part of the product is designed than in waterfall and the implementation follows the design more closely.</td>
</tr>
<tr>
<td>5</td>
<td>Most important design is worked on first and no effort is wasted on specifying features that will not be built.</td>
</tr>
<tr>
<td>6</td>
<td>Developers can maximize programming time as they don't have to wait for design deliverables to begin their work.</td>
</tr>
<tr>
<td>7</td>
<td>Development effort is not wasted on programming things that eventually would turn out to be wrong as requirements and design concepts are elaborated up-front (as opposed to agile).</td>
</tr>
<tr>
<td>8</td>
<td>Usability investigations are conducted throughout the entire project.</td>
</tr>
<tr>
<td>9</td>
<td>The concept of time-boxed sprints helps developers to focus features that are important to users and to test with the related specific set of user tasks.</td>
</tr>
<tr>
<td>10</td>
<td>Changes suggested by usability testing (and other methods) can be implemented in the current release.</td>
</tr>
<tr>
<td>11</td>
<td>Quality assurance work can benefit by including usability testing success to acceptance criteria of features.</td>
</tr>
<tr>
<td>12</td>
<td>Timely feedback is always available.</td>
</tr>
</tbody>
</table>

6.2. Integrated life-cycle

UCD work within the integrated systems development process can be divided into four groups (Nodder & Nielsen, 2009, p. 65):

1. early work, e.g., site visits and other user research that happens before the actual implementation starts,
2. sprint-specific work, e.g., detailed UI design and usability evaluations,
3. decoupled work or work that happens outside sprints, e.g., additional research or stakeholder workshops, and
4. post-sprint work, e.g., usability validation or UI redesign and refactoring.

As this division suggests, UCD work should have an impact throughout the development life-cycle as illustrated in Figure 9. Early on it can ensure that the correct tone is set, focus is placed on the right users and their important tasks, and user stories are written with a true understanding of the users' goals. Up-front work (in sprint 0 and before) collects valuable data for the requirements development process. Prototyping with low-fidelity wireframes is used to flesh out the requirements (used also for
usability testing), but are gradually replaced by the working software. The UCD specialist supports the team as the interface is being coded, but should also be looking into future sprints. (Nodder & Nielsen, 2009, pp. 43-44)

The big question is that should sprint level interaction design be done just in time (within the ongoing sprint) or a sprint or two up-front? This again depends on the project. Projects with well defined style guides, UI development frameworks, standard interactions and adequate resources could manage to design the necessary pieces of the UI within a sprint, just 'moments' before development. Projects with novel or complex interactions and an undefined setting for UI design and development have to take the time to do the work up-front. Another important question is whether the up-front user research should be done within the development project or should it have a life of its own outside the actual development process? Norman (2006) argues it is too late to do user research once a product development project has been announced. The information produced by the research is needed to decide what product to build and which projects to fund. Researching and developing personas outside the individual development projects could be an effective strategy to create a pool of information about the users, which could then serve the entire organization and family of products.

One solution to these questions is parallel track development (Miller, 2005; Sy, 2007). In that approach the UCD specialists divide (and estimate) their work in a similar manner as the developers do. Work breakdown is done by features and not by tasks, and these design chunks must be small enough to be completed in a single sprint. While this division of work can be hard to do at first, as UCD specialists are used to thinking

Figure 9: UCD work in Agile development cycle. Modified after Nodder & Nielsen (2009, p. 43).
about systems holistically, it allows the whole user interface design and evaluation to be
done in small increments over several sprints (Sy, 2007). The parallel track process is
illustrated in Figure 10.

In parallel track development, UCD work is sequenced in the following way
(Nodder & Nielsen, 2009, p. 57):

- User research (for cycle n) is done in cycle n-2.
- UI design (for cycle n) is done in cycle n-1.
- Development is supported in cycle n.
- UI (from cycle n) is evaluated in cycle n+1.

In other words, as described in Figure 11, up-front user research (the creation of
personas) and requirements gathering are done in Sprint 0 (and before), prior to the
actual programming. Sprint 1 starts the programming, during which the developers
intentionally focus on features that are not UI intensive. This buys time for UI design
work the UCD specialists are doing in parallel for Sprint 2. They also gather data for
Sprint 3. In Sprint 2 the UCD specialists share the design with the developers, design
for the next sprint and gather data for the sprint after that. They also start usability
testing work from the previous sprint. The cycle repeats itself through the project.
(Nodder & Nielsen, 2009, p. 57)
According to Nodder & Nielsen (2009), the largest concern in parallel track development is that by separating design from implementation the development moves away from the “one team equality model” (Nodder & Nielsen, 2009, p. 59). Teams that are not communicating properly (and perhaps not co-located) and are new to agile philosophy could degrade back to the old, bad ways of working where decisions are just thrown over the wall without much collaboration between disciplines.

Fortunately the team can mitigate this problem by setting up good communication channels and supporting each other's work which is required for the parallel track development to work. Sprint 0 should be used effectively for team building and sharing any research results and design already done to align team members’ understanding of the project (Fox, Sillito, & Maurer, 2008).

In fact, as Ramsey (2010) suggests, it's not the UCD specialist who should be designing ahead but the whole team together (facilitated by the UCD specialist). Patton (2003; 2004) gives a good example of a technique based on Usage-Centered Design where business stakeholders, users and developers all work together to produce the initial design solution. Design studio is an other technique that could facilitate collaborative design by “focusing the team's collective energy and ideas on design direction” (Ungar & White, 2008). The idea is that each team member individually produces design sketches based on user research. The sketches are then discussed together and merged in to one design concept, which is then used to move forward in the development (Ungar & White, 2008). The technique sounds promising. The product vision and design are communicated between the team, and it offers a good opportunity to share user research findings, promote UCD in general and build the team cohesion. Design studios and other collaborative design (and development) practices are key tools in the integration effort.

Figure 11: Integrated life-cycle. Modified after Nodder & Nielsen (2009, p. 43, 57).
Besides communication and team building issues, projects can suffer from various other kinds of problems while the integration is being done. Table 10 summarizes those problems (and possible solutions) that Sy and Miller (2008) have experienced in their work or collected from discussions with other UCD practitioners.

### Table 10: Problems experienced by UCD practitioners while doing agile UCD.

Modified after Sy & Miller (2008).

<table>
<thead>
<tr>
<th>#</th>
<th>Problem</th>
<th>Symptoms</th>
<th>Possible solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not enough design time</td>
<td>• Developers are waiting on designs.</td>
<td>• Develop in separate and parallel UCD Design/Developer tracks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Design quality drop.</td>
<td>• Scope UCD activities to be small and incremental.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Designs not verified with customers.</td>
<td>• Use RITE usability testing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Do rapid contextual design.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Do design chunking.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Combine different UCD activities into one session.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Bring user to you.</td>
</tr>
<tr>
<td>2</td>
<td>Sprints are too short</td>
<td>• Designs can’t be finished in time.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No time for usability testing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No time to set up customer contact.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Not enough user feedback</td>
<td>• Feedback is not early enough.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No data to act on - opinions rule.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Product isn’t validated.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Weak agile customer</td>
<td>• End-users and clients won’t participate.</td>
<td>• UCD person can act as agile customer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Can’t get buy-in from rest of team.</td>
<td>• Each UCD person works on one scrum team.</td>
</tr>
<tr>
<td>5</td>
<td>UCD is not full-time on one agile team</td>
<td>• UCD time spent in many meetings. instead of on designs and iterations.</td>
<td>• Choose the scrum teams wisely.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demoralized by UCD quality drop.</td>
<td>• Validated designs are passed to developers to implement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• UCD participates in cycle planning bringing appropriate user feedback.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• No features go in unless something comes out.</td>
</tr>
<tr>
<td>6</td>
<td>No sprint planning</td>
<td>• Large backlog of features/bugs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Prioritization feedback is ignored.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No control over timing of designs.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>User feedback is ignored</td>
<td>• Feature set is cast in stone.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No time to incorporate changes.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No re-ordering of features is allowed.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Missing the ‘big picture’</td>
<td>• No shared vision or end goal.</td>
<td>• Persuade the agile team to adopt Sprint 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Too much focus on details.</td>
<td>• Lighten the requirements gathering process.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hard to prioritize/make design decisions.</td>
<td>• Shorten time to 1 or 2 sprint lengths.</td>
</tr>
<tr>
<td>9</td>
<td>Poor communication</td>
<td>• Misunderstood designs.</td>
<td>• Include developers in design process.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Agile team doesn’t buy-in designs.</td>
<td>• Include usability in acceptance criteria.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Important information is lost.</td>
<td>• Contact developers daily to check progress.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Use design cards for stand-up meetings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Use issue cards for usability reporting.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Use documents internally for the design team.</td>
</tr>
<tr>
<td>10</td>
<td>Team isn't co-located</td>
<td>• No sense of team - lack of trust.</td>
<td>• Use telecommuting tools (phone and web-based replacements).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Language and/or time barriers.</td>
<td>• Co-locate for cycle planning.</td>
</tr>
<tr>
<td>11</td>
<td>Dependency issues</td>
<td>• Requiring input from non-agile teams (e.g., marketing sign-offs, lawyers).</td>
<td>• A scrum leader or facilitator with strong persuasion skills can move things along quickly.</td>
</tr>
</tbody>
</table>
Project management or the resourcing of UCD work are not directly mentioned in Table 10, although the risk of overburdened UCD specialists is evident while the integration is taking its initial steps. Learning new practices and adapting old ones and educating others about UCD, while trying to provide the developers with necessary designs in order for them to continue working, can seem a daunting effort.

6.3. Project management of UCD work in the integrated process

The integration does not only change how design and development are done in a project, but also raises some considerations regarding how the work is managed. Traditionally the need for UCD resources has been heavy in the beginning of the project, but in the integrated model UCD specialists are needed through the whole life-cycle and often this involvement is a full-time role. Because of this commitment, resources can't be moved from team to another so easily. This allows UCD to integrate deeper into the development practices of the team, but can be problematic if there are too few UCD resources to begin with. Scarce resources should be allocated to those projects that are important business-wise and where UCD work will benefit the most. Although organizationally centralized UCD team is a key practice, the UCD specialist's first responsibility is to the project team. (Nodder & Nielsen, 2009, p. 37)

Regardless whether the UCD specialists are part of the development teams or move from development team to another, “a cohesive project management framework” (Chamberlain et al., 2006) is needed that facilitates the both disciplines. For example, a large development organization could be organized such that the UCD specialists form their own Scrum team which then serves the actual programming teams (Nodder & Nielsen, 2009, p. 41). In this organizational setting UCD work would be managed and tracked in its own backlog by the UCD product owner. This approach could also solve some of the problems related to tracking UCD work within user stories.

Any tasks needed to implement a user story should be included in its work estimate. Thus any UCD work should also be included in the user story to track its and the team's progress accordingly (Nodder & Nielsen, 2009, p. 59). This suggestion could work out if UCD work is done within the same sprint, but in the case of parallel track development, the approach can lead to problems. A user story can't be considered complete until all the work it involves is done and the functionality is tested to be working as intended. Including the UCD work in the user stories could mean for example that a story isn't complete until it is tested and fixed for usability errors. Thus in parallel track development a story could be 'open' for several sprints which would be problematic for tracking progress (or velocity in agile terms) (Nodder & Nielsen, 2009, p. 105). A probable solution would be that UCD work that happens before and after the sprint is tracked as separate 'usability stories' and the UI implementation support that happens within the sprint is included in the actual user story. This way the team can
show early progress as stories are completed in every sprint and the UCD specialists are not overwhelmed by work as it is estimated in the same manner as everything else.

6.4. Best practices to agile UCD

With the increased popularity and adoption of agile development in the industry, the UCD practitioners have had to re-think how they fit into the new processes. A set of best practices to agile UCD have began to emerge (Patton, 2008) from this thinking. The ideas and experiences from agile UCD pioneers have spread and are being put to practice and improved by others in the industry around the world.

Table 11 is a collection of these emerging best practices. More specifically it is a collection of practices on how to fit and do UCD work within an agile process. It does not contain practices that deal with integration principles or organizational adoption. Each practice has been discussed earlier in Chapters 5 and 6 and in various sources like (Chamberlain et al., 2006; Miller, 2005; Nodder & Nielsen, 2009; Patton, 2008; Sy, 2007).

Table 11: Agile UCD best practices.

<table>
<thead>
<tr>
<th>#</th>
<th>Practice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UCD must adopt the agile mindset</td>
<td>UCD specialists must adopt the agile mindset before the changing of working habits and practices from waterfall UCD to agile UCD can succeed. UCD specialists must view themselves as members of the development team first and UCD specialist as second.</td>
</tr>
<tr>
<td>2</td>
<td>UCD should be close to product owner</td>
<td>UCD specialists should work closely with product owners (or in some cases be the product owner). This way they can have influence on what is going to be built and what are the goals for the software, and ensure that end users are taken into account when making these decisions.</td>
</tr>
<tr>
<td>3</td>
<td>UCD should be co-located</td>
<td>The UCD specialists should be co-located with the rest of the development team to enable good communications and education of UCD practices to developers, and to build a truly cross-functional team. Tacit knowledge about the software, which is gained through collaboration, compensates the lack of extensive specifications.</td>
</tr>
<tr>
<td>4</td>
<td>Cross-functional team</td>
<td>Every team member participates in each step and activity in the life-cycle. The whole team designs together (e.g., design studios), UCD specialist facilitating the collaborative design effort. This educates developers about UCD and helps to share research findings and knowledge about the design.</td>
</tr>
<tr>
<td>5</td>
<td>Common project management</td>
<td>Both agile &amp; UCD should have a common setting for project management. UCD work is broken down by feature (design chunking) and estimated and its velocity tracked as story points. This makes the overall project management effort simpler and keeps the UCD specialists from being over burdened with work.</td>
</tr>
<tr>
<td>6</td>
<td>Design partner program</td>
<td>A program aiming to provide enough participants to user and usability studies on a steady stream. Some end-user representatives can become &quot;partners&quot; by committing to help for the duration of the whole project, while others participate in a single study. Ongoing recruitment of end-user representatives makes logistics easier and enables revolving door usability studies.</td>
</tr>
<tr>
<td>7</td>
<td>Up-front UCD work</td>
<td>Some UCD work such as user research, high level requirements, concept design and initial UI design should be done before development begins. Some of this work is done outside the project, some in Sprint 0 and the rest in Sprint 1, if time is bought for the UCD work by letting the development begin from non UI-intensive features. This up-front work reduces the risk of building the wrong thing and wasting resources while doing that.</td>
</tr>
</tbody>
</table>
De-coupled user research

Some user research should be independent from the actual project and run continuously as a separate program to serve the whole development organization.

Design chunking & just in time design

The design is broken down into smaller 'chunks' and done incrementally just like the requirements/features are broken down to user stories and constructed over several sprints. This enables common project management for both design and development work. Chunks of detailed UI design are done just in time before needed either during the same sprint or one before development so that resources are not wasted on designing something that will not be built.

Parallel track development

UCD specialists work parallel to development, but sprint or two ahead. In a given sprint they do user research for two sprints ahead, UI design for one sprint ahead, support developers for the current sprint and do usability evaluations for the previous sprint. This buys enough time to do necessary UCD work during the development.

UCD enhanced user stories

UCD tasks should be included in user stories to properly estimate the amount of work. The acceptance criteria or test for the story should include usability requirements. In some cases a separate usability story should be written, e.g., for usability issues or work that only includes UCD tasks allowing their prioritization and tracking.

Solution focused usability testing

Usability testing should be started early and done repeatedly through the project. It should focus on helping the team to effectively find and solve usability issues (e.g., with RITE). Testing and its logistics should be as lightweight as possible and it should be done regularly (e.g., revolving door studies).

There are dependencies between the best practices presented here. For example, design chunking makes common project management possible as both UCD and development work are broken down and estimated in the same way. This in turn helps to prioritize, choose and schedule UCD work to be done before programming which is needed to feed developers with the necessary designs in a parallel track.

Agile teams should be empowered to choose the work practices they see fit to use in a project. If the team manages to do just enough UCD work within the sprint to continue working effectively, parallel track development is not needed. It is often the development teams and single practitioners that start the organizational adoption of new practices and methodologies. They have a lot to say how UCD eventually integrates into agile development.
7. How to adopt agile UCD in an organization

7.1. Incentives for process improvement

When an organization considers adopting an agile methodology it is important to remember that Scrum and XP (or any other) alone might not be enough to describe and solve all the issues in the product development life-cycle. Small organizations and teams are probably more than satisfied with what these methodologies have to offer. Large organizations on the other hand need a delivery framework that comprehends all the phases and levels of product development all the way from the portfolio to the daily cycle. Also agile methodologies cannot be adopted by thinking that only the software development process needs to change in order to gain the potential benefits. If the delivery framework (and the organization) around the software development process does not change and become agile too, it effectively becomes a bottleneck for the business (Leffingwell, 2007, p. 290; Sliger & Broderick, 2008, p. 289).

Integrating agile development with UCD creates a systems development process which brings the organization one step closer to a comprehensive product delivery framework. While the benefits of UCD are clear and it has been around for more than two decades, it still is not as widely adopted as one would imagine. Even when the term user experience has become one of the buzzwords of the corporate world, many organizations fail to see UCD activities as an investment in product development and regard it as an operational cost instead (Zapolski & Braiterman, 2003). Still many organizations have become to realize the benefits of involving end-users in product development. These organizations have had incentives for process development and usually pain is a good one as resources are more easily dedicated to things that fail rather than to the improvement of things that already work (Elssamadisy, 2008, p. 29). Nielsen writes that “people have to want to change before there's a chance of helping them to do so” (Nielsen, 2006a), thus management will be ready to consider UCD after the organization has been hurt enough by its negative attitudes against usability. There might have been a serious project failure, the organization wants to answer market demands with better products or it is just addressing general business goal like low productivity etc. The reasons might also be internal, e.g., there is a powerful advocate for UCD, personnel have been educated in UCD or perhaps UCD offers objective means for resolving design related conflicts in the development team (Mayhew, 1999, pp. 412-414).

Besides dividing the incentives into to external and internal, they can be divided into business problems and process problems (or smells as they are often called in the agile community). Process problems are generally easier to identify than business
problems, but they should not be the primary drivers for adoption as they are not directly related to business value (Elssamadisy, 2008, p. 32). Table 12 displays a collection of these problems which are common reasons to trigger process improvement in a software development organization.

Table 12: Problems as incentives for process improvement (Elssamadisy, 2008, pp. 30-35).

<table>
<thead>
<tr>
<th>Business problems</th>
<th>Process problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Quality delivered to customer is unacceptable and affects negatively to revenue.</td>
<td>6. Inadequate customer involvement which leads to the business problem of non-useful software.</td>
</tr>
<tr>
<td>2. Delivering new features to customer takes too long and can't keep up with the competition.</td>
<td>7. Management has little visibility into the real progress of project's development. Possible budget or schedule overruns are identified too late.</td>
</tr>
<tr>
<td>3. Some features are not used by the customer or the software is not useful at all because it doesn't meet what is actually needed.</td>
<td>8. Key resources become bottlenecks in the progress as they are assigned to several projects concurrently. Multitasking between the projects reduces productivity.</td>
</tr>
<tr>
<td>4. Software is too expensive to build causing competitors to win all the work.</td>
<td>9. Work is wasted by reporting hundreds or thousands of bugs in to an issue tracking system when only the most critical are addressed.</td>
</tr>
<tr>
<td>5. 'Us versus them', e.g., the customer and the development team don't work together for the same business objective but treat each other as opponents.</td>
<td>10. Before releasing the software, a “hardening” phase/sprint is needed to fix bugs and do additional testing which is an indication that the actual development sprints were not done properly.</td>
</tr>
<tr>
<td></td>
<td>11. Integration is infrequent and not done in every sprint. This results in a situation were a lot of the errors, miscommunication and misconceptions of the software go undiscovered until the integration is done.</td>
</tr>
</tbody>
</table>

The VersionOne survey (VersionOne survey, 2008) lists several reasons why teams and organizations have adopted agile methodologies. Table 13 lists the top 8 reasons from the survey (together with their frequency in the VersionOne data set). According to my analysis they are directly related to the problems listed in Table 12.

Table 13: Reasons to adopt agile methodologies and their corresponding problems in Table 12.

<table>
<thead>
<tr>
<th>“Most important reason for adopting agile”</th>
<th>% of answers</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerate time-to-market</td>
<td>22%</td>
<td>2, 8</td>
</tr>
<tr>
<td>Enhance ability to manage changing priorities</td>
<td>21%</td>
<td>3</td>
</tr>
<tr>
<td>Increase productivity</td>
<td>12%</td>
<td>4, 8, 9</td>
</tr>
<tr>
<td>Enhance software quality</td>
<td>10%</td>
<td>1, 10, 11</td>
</tr>
<tr>
<td>Improve alignment between IT and business</td>
<td>9%</td>
<td>5, 6</td>
</tr>
<tr>
<td>Improve project visibility</td>
<td>6%</td>
<td>7</td>
</tr>
<tr>
<td>Reduce risk</td>
<td>6%</td>
<td>7, 11</td>
</tr>
<tr>
<td>Simplify development process</td>
<td>4%</td>
<td>11</td>
</tr>
</tbody>
</table>
While the incentives for adopting agile development or UCD are not inherently different, the setting for the adoption can be quite different. When an organization adopts agile development it adopts a software development methodology, whereas UCD is a design process that is intended to be integrated into a methodology to complement it. Usually in a software development organization there are many software developers and few UCD specialists. Potentially a much larger percentage of the people in the organization is familiar with agile development when the adoption begins than with UCD. Although the developers and UCD specialists should be equal brothers-in-arms in the integrated process, the UCD specialists still need to work a lot to get in to that position.

7.2. Considerations regarding agile adoption

Quite often the agile adoption is not initiated by the organization as such (even with the right incentives in place) but begins spontaneously from bottom-up as a single team starts using one or several agile practices. When the team shows good results, by implementing Scrum for example, word gets around and other teams try it out as well. Because agile adoption requires changes both in the software development process and in the surrounding functions and organization, it is crucial to have an executive sponsor for the adoption program (Leffingwell, 2007, pp. 295-296). In the VersionOne survey (VersionOne survey, 2008) the following reasons were listed as the top four barriers for further agile adoption:

1. inability to change organizational culture,
2. general resistance to change,
3. no personnel with the necessary agile experience, and
4. lack of management support.

Without the support of an executive champion acting as a change agent, the adoption might take too long (Leffingwell, 2007, p. 301) or the benefits gained might stay local and not spread through the organization. The adoption might fail altogether due to the reasons listed above and the teams slip back to their old ways of working. Leffingwell (2007, p. 301) suggests that executive management should not only respond to the change started from bottom-up, but that it should be lead top-down and driven as a Scrum project, where barriers for adoption are tackled iteratively.

After the organization has made the decision to go for agile, it needs a strategy and a plan for the adoption program. Elssamadisy (2008, p. 49) suggests three different approaches to deciding about the actions to be taken and agile practices to be adopted:

1. Choose practices based solely on business value. In this scenario the organization is not suffering from severe problems but wants to improve the development process by increasing the business value that teams deliver.
2. Choose practices to alleviate smells that have been prioritized by business value. In this scenario the focus is on fixing problems the organization has prioritized by business value.

3. Choose practices to address the most visible smells. In this common scenario problems are addressed starting from the biggest but not considering how much business value will be delivered by fixing each.

Elssamadisy (2008) has also created a framework for choosing the practices to be adopted based on the approaches above. In this framework an appropriate set of practices is chosen according to the identified area of improvement and then adopted iteratively as follows:

1. Current situation is evaluated and areas of improvement are measured.
2. Goals are set for how much the measurements should improve in a given time frame.
3. The most effective practice is picked from the chosen set and taken into use.
4. Periodical evaluation is done whether the usage of the practice is causing the desired effect and if not, the usage is adjusted.
5. Situation is re-evaluated and if the set goals are not met, a second practice is taken into use. If goals are met then the adoption program moves on to a new issue.

The Unified Process (UP) framework also recommends an iterative, incremental and adaptive adoption program for itself. The executive management is trained first and an executive sponsor is chosen for the program. A suitable pilot project is chosen which should be meaningful and interesting enough but which doesn't involve high risks and is likely to succeed. An UP coach (experienced methodology expert) should lead the process and coach the project team about the practices. After the project is successfully completed the team then actively promotes UP in the organization and ideally some of them will act as the 'process engineers' or coaches in the second generation of UP projects. (Larman, 2003, pp. 203-204)

According to Blomkvist (2005), the idea behind the Agile Manifesto is that there is no single development process to fit all purposes and that process customization is needed to fill the needs of different projects. Interestingly this kind of methodology tailoring during the adoption is quite contrary to what Scrum and XP suggest. Both recommend that the most difficult or worst project or problem should be chosen as for the pilot and all the practices should be taken into use as described (Larman, 2003, pp. 132, 165). This is especially true for XP as the practices create synergy by supporting each other and it can be 'dangerous' not to adopt all of them before becoming experienced in XP (Larman, 2003, p. 165). Sliger & Broderick also have the same view:
You can implement individual practices here and there, but they will not give you the same return as if you implemented all the pieces that make Agile what it is. All of the pieces and parts of Agile work together to make one big working system. (Sliger & Broderick, 2008, p. 288)

To get the maximum benefits from any new methodology, all the related software development practices should be changed at once. If the changes are small or simple this might work, but for most situations agile practices need to be introduced slowly one by one for an existing organization. It takes time to get enough experience in the new practices and how the development process works. If all practices are changed at once, the development organization might not work effectively enough during the adoption. The risk of issuing an all-out, full scale adoption program might be too high. So in theory a wholesale adoption is a good thing, but it rarely succeeds in practice. (Aguanno, 2004, pp. 338-339)

There is no reason why partial agile implementations should fail. It is not important to adopt all practices at once, but to set the right expectations through constant communication. The deciding factor in agile project's success is “whether the team can get creative, picking the parts of Agile which make them better, reflecting often what is going right and what needs fixing, and then making the course corrections to ensure constant improvement” (Nodder & Nielsen, 2009, p. 26).

7.3. Considerations regarding UCD adoption

According to Nielsen (2006a; 2006b) it can take even up to 20 years for a large organization to develop mature UCD processes. During this time the organization progresses through the following stages of maturity:

1. hostile towards usability,
2. developer-centered usability (usability is cared for, but not practiced),
3. skunkworks usability (usability testing is conducted in some projects),
4. dedicated usability budget (usability is planned as a process),
5. managed usability (dedicated usability team),
6. systematic usability process (from “just usability testing” to full UCD),
7. integrated User-Centered Design (research and data centric UCD-process), and
8. user-driven corporation.

People and organizations are inherently resistant to change. Often UCD needs to be sold to organizations that are not likely to be interested in it in the first place. One of the most important skills of the UCD professional is the ability to communicate the UCD message effectively throughout the entire organization (Six, 2009). Once it has been established that the organization is not outright hostile towards UCD, actions can
be taken to adopt it. Nielsen (1993, p. 21) recommends the following list of five action points to get started on a systematic approach to usability in an organization:

1. Recognize the need for usability.
2. Make it clear that usability has management support.
3. Devote specific resources to usability engineering.
4. Integrate systematic usability engineering activities into various stages of your development life-cycle.
5. Make sure that all user interfaces are subjected to user testing.

As the organization starts to care for usability and recognizes its need, the situation needs to be assessed before UCD can be sold effectively. What is the organization's true readiness to accept UCD and where can it have the best effect? A usability champion is needed to make the assessment. Although there are many frameworks for usability maturity evaluation (Jokela, 2010, p. 79), the champion could use a simple list of questions in Table 14 to do the assessment.

Table 14: List of questions for assessing organization's readiness to accept UCD (Six, 2009).

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>How are products designed and developed today? Where can UCD integrate?</td>
</tr>
<tr>
<td>2</td>
<td>What is the company vision? Does it use the right words that make it receptive to a UCD sell?</td>
</tr>
<tr>
<td>3</td>
<td>Who is working on design today? Where does it sit in the organization and who owns it?</td>
</tr>
<tr>
<td>4</td>
<td>Is there anyone at the strategic level championing UCD currently? This is important.</td>
</tr>
<tr>
<td>5</td>
<td>Is there anyone at the product or project level championing UCD currently? This is just as important as #4.</td>
</tr>
<tr>
<td>6</td>
<td>What are their high profile products and services? What are they doing well? How could UCD help? What UCD learnings are there? How could you use these as stories that tell why UCD is a good thing?</td>
</tr>
<tr>
<td>7</td>
<td>How can the teams be helped to work better toward meeting a UCD vision?</td>
</tr>
<tr>
<td>8</td>
<td>What does the company know about their customers today? How do they know it? How can you help them learn more? How can you compliment their current understanding?</td>
</tr>
<tr>
<td>9</td>
<td>What type of culture exists now? Is the organization engineering-centric, design-centric, sales-and-marketing-centric, or something else?</td>
</tr>
</tbody>
</table>

Besides making the assessment, the role of the usability champion is to drive the adoption program forward. Rubin and Chisnell (2008) define the champion as “someone within management with influence, one who is committed to the philosophy of usability and has a personal interest in seeing such a program blossom and flourish” (Rubin & Chisnell, 2008, p. 317). In practice two types of champions are needed. One that is a UCD practitioner and who drives the adoption program in practice (Gulliksen et al., 2003). The other one is from management that backs up and supports the effort and who's role is to clear organizational obstacles for the adoption to proceed. In agile adoption the whole pioneering development team is the champion, whereas the UCD
adoption often rests on the shoulders of one practitioner. Thus the champions might need support from peers, outside consultants etc.

Mayhew (1999) writes that those who are “pursuing the goal of introducing usability engineering techniques and practices into a development organization for the first time” have to view themselves first and foremost as change agents rather than usability specialists (or any other role). Not taking this view will most likely result in failure to integrate UCD into the software development process in a lasting way. The key is to understand what motivates organizations to change. The obstacles to adopting UCD can be divided into attitudes against it as well as issues in organizational incentives, practices and structures which must be identified, analyzed and dealt with. The change agent has different factors, approaches, techniques, tactics and strategies that are used to address obstacles and that “can serve as facilitators in effecting change”. The facilitators are either motivators that create opportunities for change, or success factors (see Table 15) whose strategic application will make the change happen. (Mayhew, 1999, pp. 404-405)

Table 15: Success factors for implementing organizational change to adopt UCD (Mayhew, 1999, pp. 415-422).

<table>
<thead>
<tr>
<th>Success factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Establish credibility</td>
<td>Establish credibility among co-workers by conducting usability engineering tasks that require specific skills.</td>
</tr>
<tr>
<td>Communicate effectively</td>
<td>UCD practitioners need to be fluent and effective communicators before credibility can be established.</td>
</tr>
<tr>
<td>Get 'buy-in'</td>
<td>Get 'buy-in' by working as a member of the development team educating the team members about UCD instead of working as a solo designer/specialist.</td>
</tr>
<tr>
<td>Be an engineer, not an artist</td>
<td>Software developers are engineers and working like a fellow engineer will help to put the language and cultural barriers aside.</td>
</tr>
<tr>
<td>Be an ally, not an enemy</td>
<td>Usability work can be seen as a critique against the product and the team. Instead, sell it as an invaluable tool for success.</td>
</tr>
<tr>
<td>Produce well-defined work products</td>
<td>Produce well-defined work products by working like an engineer and not just throwing in design opinions.</td>
</tr>
<tr>
<td>Manage expectations</td>
<td>Manage expectations by clearly stating which issues you can help with and what are the limitations of your work. Credibility can be seriously damaged when (unrealistic) expectations are not met.</td>
</tr>
<tr>
<td>Clarify 'value-added'</td>
<td>Explain what the actual potential benefits of UCD might be for the particular product and choose a highly visible project in which those usability engineering tasks are applied that will prove the value of UCD quickly and dramatically.</td>
</tr>
<tr>
<td>Test whenever possible</td>
<td>Test whenever possible. Usability testing is a strategic political tool as the data it produces can be quite convincing in selling UCD.</td>
</tr>
</tbody>
</table>

Mayhew (1999) continues that in an organization inexperienced in UCD, it is impossible to convince it to try and then to successfully implement the whole Usability Engineering Lifecycle (see Figure 4) at once. Instead it is better to take small steps and introduce usability engineering tasks one at a time, one project at a time to gradually
evolve the organization towards integrating the whole life-cycle into their development process. For example, it is easy to start doing usability testing late in the project life-cycle. This almost always demonstrates the value of UCD techniques and wins the support and funding for more usability work (although the impact to the usability of the product being developed might be low). Then it is easier to move into UCD tasks earlier in the life-cycle that have a higher impact on the usability of the products. The adoption of UCD should begin from that section or business unit in the organization that is more receptive and from those projects that have high visibility and impact on the organization. (Mayhew, 1999, pp. 423-424)

Rubin and Chisnell (2008) also suggest making the adoption gradually in a phased program that extends over several years. Implementing user-centered approach to product development for organizations is a major undertaking that involves the same issues as any major shift in the corporate culture. Depending on the resources and support this change might happen faster or slower. The adoption program tends to be more organic rather than formally planned and there is no single recipe which would suit the needs of every organization. Pilot projects should be chosen carefully from those that have high visibility in the organization and that are also likely to succeed. (Rubin & Chisnell, 2008, pp. 315-317)

It is important to build a usability program slowly, success upon success, rather than suddenly trying to reinvent the organization. Usability, like its counterpart, quality, cannot simply be mandated from above. It needs the support of everyone and in the organization so that it can enter the organization's bloodstream, rather than existing as some superficial 'user-friendly' flag that everyone waves, but that produces very little results. (Rubin & Chisnell, 2008, p. 320)

The adoption program should have a plan for execution that extends over years and that integrates into the business goals and the long term vision of the organization. The plan should have checkpoints and key performance indicators (KPIs) to measure the progress of the program and to realign it to match the changing business goals along the way.

The following process model (see Table 16) divides organization's UCD maturity into four phases in a ten year timeline. It also describes steps that can be taken to achieve each phase of maturity. The model could be used as a basis for planning the UCD adoption program.
Table 16: A process for adopting UCD (Rubin & Chisnell, 2008, p. 316).

<table>
<thead>
<tr>
<th>Year 1-2 Stealth Mode: Establish Value</th>
<th>Year 2-4 Build on Successes</th>
<th>Year 3-6 Formalize Processes and Practices</th>
<th>Year 4-10 Expand UCD Through the Organizational Practices</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Status</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-2 visible tests with publicized results.</td>
<td>Usability tests are scheduled a few times a year.</td>
<td>The organization knows that there are usability specialists in the company.</td>
<td>The organization values User-Centered Design and strives to attend to the overall customer experience.</td>
</tr>
<tr>
<td>1 person doing usability tests, 1 champion, using surrogates or &quot;friends and family&quot; for participants.</td>
<td>Usability resources may be overextended as demand grows.</td>
<td>A few product groups do usability testing regularly.</td>
<td>User research and usability activities are assumed to be part of every development project.</td>
</tr>
<tr>
<td>Probably no usability budget: money is scrounged or &quot;donated&quot; for participant incentives and other costs.</td>
<td>Small budget for overhead: may have set up a charge-back system within the company to fund test and other projects.</td>
<td>Budget covers 2-3 staff, dedicated equipment and possibility space.</td>
<td>Methods are combined to reveal the richest, most realistic data.</td>
</tr>
<tr>
<td></td>
<td>Scrounging for space to use as a center usability area.</td>
<td>Methods expand from usability testing to include field research for user and task analysis.</td>
<td>Funding and projects are coordinated with R&amp;D, market research and others.</td>
</tr>
<tr>
<td><strong>Next Steps</strong></td>
<td>Set up a long term relationships with as many product groups as you can support and do research for them directly.</td>
<td>Set up central residency.</td>
<td>Get more education.</td>
</tr>
<tr>
<td>Make friends in the company: get buy-in to new products.</td>
<td>Gather more champions and friends.</td>
<td>Begin working with product teams and IT to build usability testing and UCD into processes throughout the product development life-cycle.</td>
<td>Standardize processes.</td>
</tr>
<tr>
<td>Talk about this cool new stuff you're doing and how it might be applied to other products and departments.</td>
<td>Sell your services internally by deliberately marketing success and training and coaching product teams to do their own research.</td>
<td>Continue training and coaching product teams.</td>
<td>Align with other groups.</td>
</tr>
<tr>
<td>Educate yourself.</td>
<td>Strategize by using real customers as participants, implementing iterated usability tests, bringing a vendor for special projects that need objectivity or specialized skill.</td>
<td>Distribute videos or podcasts about your findings and successes.</td>
<td>Implement more research methods, expanding the scope and reach of research.</td>
</tr>
<tr>
<td>Volunteer your services.</td>
<td>Create a budget based on a business case for User-Centered Design.</td>
<td>Expand your business case.</td>
<td>Evaluate your efforts.</td>
</tr>
<tr>
<td>Create a strategy and business case.</td>
<td>Set up standing dates for doing usability testing in a central location and invite observers.</td>
<td>Increase your budget request.</td>
<td>Expand your UCD reach beyond priority products to looking at the larger customer experience of researching, buying, installing, using, troubleshooting etc. or from venue to venue (e.g., from a website to retail site). Add more researchers, designers, prototypers, recruiters that form an interdisciplinary group, increase your budget and revise your business case, organize internal conferences on UCD projects, practices and outcomes and keep making friends.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add another usability specialist, an interaction designer and a recruiter.</td>
<td>Look for groups to start coming to you for services and advice.</td>
</tr>
</tbody>
</table>
Whether the organization is adopting UCD, agile development or both or is trying to integrate them, the day-to-day activities towards this process improvement are executed by the teams and individuals doing the actual software development (who probably started it in the first place). For the effort to succeed, it needs a long-term plan which is not only supported by the management but actively led from top-down. Champions are needed both among the practitioners and management.

While it might be tempting to make a complete overhaul to the development process, it is unlikely that the organization can take the necessary time off from existing projects to pull such a feat at once. So while an all-out adoption is better in theory, a program where practices are taken into use gradually over a long period of time is more likely to succeed in practice. Even when the organization is doing a gradual adoption, it is recommendable that the methods and practices should be tailored only with great consideration until the teams have gained enough experience using them out-of-the-box.

The adoption program also needs KPIs that can be used to get and retain the necessary management support (cost-justification), to determine if the teams are actually doing better or not, how the adoption is succeeding at large and where to focus next. Thus the KPIs and their analysis needs to provide actionable insights (Kaushik, 2007, p. 15).
8. Conclusions

Some issues in traditional software development, e.g., the lack of user input and incomplete and changing requirements and specifications, can cause major problems to project schedules, budgets and quality. Regarding software development as traditional engineering plus predictable manufacturing is one of the leading causes behind these issues. Instead, software development should be considered to be new product development, where very little is certain and requirements change often.

Both UCD and agile development are approaches to software product development that can effectively increase the chances that a project will be a success. UCD is a design process concentrating on user research, user interface design and usability evaluation and it is meant to be integrated into any existing software development process. Agile on the other hand refers to a number of iterative and incremental software development methodologies that emphasize people, communication and the ability to adapt to change rather than the process, tools and predictive planning. Scrum and XP, the most notable of these methodologies, are often used together to complement each other.

Integrating UCD into such agile development framework creates a comprehensive systems development process. While there are no insurmountable obstacles in integrating them, both principles and practices of these two approaches need to be reconsidered. Agile development can't be considered to be user-centered – there is little time nor practices for doing user research, up-front design and usability testing. UCD on the other hand is not incremental and its practices can be too heavy for agile settings. Both need to change for the integration to succeed.

The solution is to include a UCD specialist in the agile project team and to buy enough time to do UCD work before programming. This is achieved by doing UCD incrementally, where it is broken down and estimated by feature in the same way as programming tasks. While some user research and initial concept design needs to be done before the programming begins (in Sprint 0 and even earlier), most of the UCD work should be done sprint or two before programming. The up-front design reduces the risk of building the wrong thing and wasting resources while doing it. Just-in-time design on the other hand ensures that resources are not wasted on features that might not be built. Thus UCD work should be done on a parallel track with programming.

In addition to on-site customers, actual end-user representatives should participate in development through user studies and early, solution focused usability testing. The UCD specialist on the project should work close to the product owner and act as bridge between users, customers and the rest of the development team. While the UCD specialist should own the user experience of the product and be responsible for it, the
team should be cross-functional, each member participating in all key activities of the development. Design should be done in collaboration facilitated by the UCD specialist. Although best practices to agile UCD have started to emerge, each integration is unique in some aspects as every project, team and organization is different. The keys for success are mutual respect between different disciplines and close collaboration and education through shared work activities.

Whether the organization is about to adopt UCD, agile development or both at the same time, the change usually starts from single practitioners and teams at the bottom. As practices and results start to spread, management should step in by supporting and leading the adoption program. The current development process should be assessed and the process improvement effort directed to those points where it can have the most effect. Change resistance and other obstacles to the adoption program should be acknowledged and managed. While in theory the greatest benefits from any new methodology or process can be gained only by adopting all its practices at once, gradual adoptions work better in practice for existing teams and organizations. The teams should be empowered to choose work practices that are the most effective for the particular project. Practices should be taken into use gradually and modified only after the team has gained experience in using them. The team should reflect those experiences often and change the process if needed. In other words, agile UCD integration and organizational adoption should be iterative and incremental – managed and executed like an agile development project.

The purpose of this thesis was to sketch the big picture of an agile UCD process and I personally feel this has been done. The next natural step for me is to see how that sketch and the agile UCD best practices found from literature will evolve when they are exposed to the realities of software development projects. Although all of the best practices reported here have been 'proven' in practice by others, each software development project is unique presenting a different context for applying the practices. This could be classified as future work, and while the sketch remains to be embellished to a painting, a solid background for such work has been provided here. I hope that other designers and developers can pick up some concrete advice from it in order to improve their software development practices for creating better and more usable software.
References


Fox, D., Sillito, J., & Maurer, F. (2008). Agile methods and user-centered design: How these two methodologies are being successfully integrated in industry.


