User-Centered Design and Extreme Programming

Antti Nummiaho

***Abstract*— User-centered design (UCD) aims at designing software so that users needs are considered throughout the process. Extreme programming (XP) is the most well-known agile software development method. It aims at being so lightweight that changing requirements even late in the project do not cause much rework. This paper focuses on identifying conflicts between UCD and XP as well as discussing ways for resolving those conflicts. The used research method is a literary research. The results indicate that the biggest conflicts arise from the fact that XP focuses on quality of the code, satisfying the customer, and designing only what is necessary at the time while UCD focuses on usability of the end product, satisfying the user, and designing the user interface properly before implementing it. The most relevant suggestions for resolving these conflicts include having UI designers and users in the development team, using a lightweight process for translating users' needs into paper prototypes, designing overall layout and look&feel upfront, and integrating usability evaluation with users into acceptance testing. The main conclusion is that only lightweight UCD practices can be included in XP. Especially usability evaluation with users lacks these kind of practices.**

***Index Terms*— usability, user-centered design, UCD, agile methods, extreme programming, XP.**

1. Introduction

THIS paper discusses the ways how user-centered design (UCD) can be taken into account when developing software using Extreme Programming (XP), which is the most well-known agile software development method. The paper has been written as part of the course T-76.5650 Software Engineering Seminar at Helsinki University of Technology in fall 2006.

1.1. Background

Traditionally software has been developed in a very inflexible way. This so-called ”waterfall model” works well if the requirements are very stable, but in many real world projects requirements tend to be quite dynamic. In the late 1990's this led to the emergence of many different agile software development methods, like for example XP, Scrum, Crystal and FDD. Basically, these methods consist of different practices that are considered to work well together in developing good quality software in the environment of changing requirements. In 2001 the founders of these different agile methods met and created a statement that defined their common principles and goals: the Agile Manifesto (Beedle et al., 2001). One of its main principles is to produce software that gives value to the customer. Meanwhile, the UCD process also evolved in the 1990's with its goals to design the software so that users' needs, wants and limitations are considered throughout the process. In other words, it aims at producing software that fits into users' world instead of forcing users to change. Soon people started to realize that these two methodologies share some similar goals and it would make sense to integrate them somehow. Many papers have discussed the issue of how UCD can be taken into account in agile methods and especially with XP. This paper tries to combine these findings into a coherent wholeness.

1.2. Research Problem

This paper tries to answer the following research questions.

1) What conflicts between UCD and XP have been identified?
2) What ways have been suggested for resolving the conflicts in order to combine UCD and XP?

1.3. Objectives

This study uses a literary research to seek answers for the research questions. The paper is mainly targeted for XP developers who are interested in incorporating a user-centered approach into their software development process.

1.4. Structure of the Paper

After this introduction chapter the paper is structured in the following way. In the second chapter definitions for usability are given and usability is discussed as a quality attribute in software development. After usability has been discussed, the principles and practices of user-centered design are introduced. In the third chapter agile methodology as well as XP and its practices are introduced. In the fourth chapter the conflicting elements of UCD and XP are identified and the reported suggestions and experiences on how UCD and XP can be combined are discussed. Finally, in the fifth chapter, conclusions of what was discussed are drawn and possible topics for further study are presented.

2. User-Centered Design

Before UCD can be introduced, it is essential to define usability. UCD's principles and practices are then examined to the extent that is sufficient for understanding the conflicting elements of UCD and XP and the suggested ways for combining them.

2.1. Defining Usability

Usability can be defined in many different ways. Three common definitions, two from ISO standards and one from Jacob Nielsen, a controversial leading authority on web usability, are given in the following.

ISO 9241-11: 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, 1988).

ISO 9126-1: The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions (ISO, 2000).

Jacob Nielsen: A quality attribute that assesses how easy user interfaces are to use (Nielsen, 1993).

These definitions emphasize different elements of usability. ISO 9241-11 emphasizes an overall goal that the software meets users' needs, ISO 9126-1 also emphasizes the detailed software design activity involved, and Nielsen emphasizes the qualitative nature of usability. It is also essential to understand, that usability as a term does not cover only one quality, but many possibly contradicting qualities. ISO 9241-11 describes usability as a combination of effectiveness, efficiency and satisfaction, ISO 9126-1 describes usability as a combination of understandability, learnability, operability and attractiveness, and Nielsen describes usability as a combination of learnability, efficiency, memorability, errors and satisfaction. These qualities are explained in more detail in the following.

ISO 9241-11 (ISO, 1988):

- Effectiveness: The capability of the component to enable tasks to be accomplished completely and without errors.
- Efficiency: The capability of the component to minimize the use of resources (persons, money, time).
- Satisfaction: The capability of the component to be pleasant to use for the user.

ISO 9126-1 (ISO, 2000):

- Understandability: The capability of the component to enable the user to understand whether the component is suitable, and how it can be used for particular tasks and conditions of use.
- Learnability: The capability of the component to enable the user to learn it.
- Operability: The capability of the component to enable the user to operate and control it.
- Attractiveness: The capability of the component to be attractive to the user.

Jacob Nielsen (Nielsen, 1993):

- Learnability: How easy is it for users to accomplish basic tasks the first time they encounter the design?
- Efficiency: Once users have learned the design, how quickly can they perform tasks?
- Memorability: When users return to the design after a period of not using it, how easily can they re-establish proficiency?
- Errors: How many errors do users make, how severe are these errors, and how easily can they recover from the errors?
- Satisfaction: How pleasant is it to use the design?

If one compares these three classifications, it seems that ISO 9126-1's understandability and learnability somewhat equal to Nielsen's learnability, but there is no correspondent in ISO 9241-11. Also, ISO 9241-11's effectiveness and efficiency somewhat equal to ISO 9126-1's operability and to Nielsen's efficiency and errors. Furthermore, ISO 9241-11's satisfaction equals to ISO 9126-1's attractiveness and to Nielsen's satisfaction. There seems to be no correspondent for Nielsen's memorability in ISO 9241-11 or ISO 9126-1.

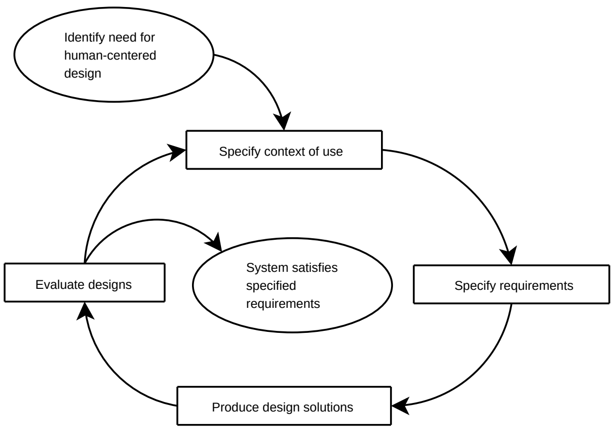

Fig. 1. ISO 13407 UCD activities.

2.2. *Usability as a Quality Attribute*

Traditionally software engineers have seen usability as something that can be added to the software after it is built (Seffah and Metzker, 2004). Therefore, the commonly used architectural decision has been to separate the user interface from the rest of the system and therefore enable its modification without affecting rest of the system. However, this is not very cost-effective since it delays the correction of problems until very late in the development process life cycle. (Bass and John, 2001). Furthermore, user interfaces can be seen to consist of both look (visual components like buttons, pull-down menus, check-boxes and colours) and feel (interaction with the user), and the latter may not be easily separated from the rest of the system (Bosch and Juristo, 2003). For example, cancel is a function that is not easily isolated from the rest of the system. It requires architectural support for recording initial states for all commands and informing impacted parts of the system when a cancel command is issued. Other architecturally significant usability issues include i.e. undo, redo, recovering from failures, and predicting task durations. Bass and John (2001) have recognized a total of 26 architecturally significant usability issues and suggested architectural mechanisms for dealing with them. Of course, architectural design must deal with other quality attributes as well, but totally dismissing usability is definitely short-sighted. (Bass and John, 2001)

2.3. *Principles and Practices of User-Centered Design*

UCD focuses on taking users' needs, wants, and limitations into account throughout the development process in order to produce software that fits into users' world instead of forcing users to change. ISO 13407 specifies four UCD activities that need to start at the earliest stages of a project: understanding and specifying the context of use, specifying the user and organisational requirements, producing design solutions, and evaluating designs against requirements (ISO, 1999). These are illustrated in figure 1.

Many approaches exist for modeling UCD. In the following a well-known model suggested by Nielsen (1993) is examined.

1) Know the user
Users are identified and their roles and tasks analysed. Potential methods include observations, interviews and questionnaires.
2) Competitive analysis
Competing products are analysed, compared, and tested.
3) Setting usability goals
Based on the analyses, it is decided how different usability attributes (learnability, efficiency, memorability, errors, and satisfaction) are emphasized.
4) Parallel design
Different design alternatives are explored before a single solution is selected.
5) Participatory design
Designs are shown to users in order to get feedback.
6) Coordinated design of the total interface
The user interface is designed making sure that it is consistent throughout.
7) Apply guidelines and heuristic analysis
Well-known principles are applied to the design and the design is checked against these principles. Heuristic analysis is the most common expert analysis method (a usability evaluation method which does not involve users).
8) Prototyping
Early prototypes are created so that design decisions can be evaluated as soon as possible. Early prototypes often take the form of paper prototypes, which enable testing at an extremely low cost. Paper prototypes may for example be hand-drawn pictures or printed screen shots that are crafted together with i.e. scissors, glue, and staples.
9) Empirical testing
The prototypes are tested with users by for example giving them some tasks and observing how they manage to accomplish them.
10) Iterative design
Based on the usability problems found in testing, new prototypes are designed, implemented, and tested. It is often too expensive to test every single improvement with users so expert analyses can be used as well.
11) Collect feedback from field use
After the product is released, feedback is gathered either passively through i.e. user complaints and calls to help lines or actively by i.e. observing and interviewing users.

The model consists of a number of specific stages for a usability engineering lifecycle. It emphasises that one should not rush straight into design, but try to understand the users, their needs, and the context first. It is important to differentiate users's needs and the functions of the user interface. For example, the user's need is not to click ”Save” on the user interface, but to be able to continue his work later on from where he left off even after the application and/or computer have been shutdown. The need could be satisfied in many other ways, but ”Save” function is commonly chosen. Therefore, it is of upmost importance to understand the users' true needs and translate them into applicable requirements. (Nielsen, 1993)

3. Extreme Programming

XP is the most well-known agile software development method. As anyone involved in software development knows, in real-world applications the requirements always change. In traditional software development this leads into a lot of documentation and design done in vain. Agile methods aim at being so lightweight that changing requirements even late in the project do not cause much rework. This is achieved by creating only the absolutely necessary documentation and design upfront. To manage this without degenerating into code-and-fix hacking and losing quality agile methods follow a set of principles and practices. Nonetheless, agile methods are best suited for small teams of skilled programmers who take responsibility for their area of expertise, and that work on non-safety-critical software. (Beck and Andres, 2004)

Agile Manifesto defines the principles for all agile methods. Most relevant principles for understanding this study are presented in the following. (Beedle et al., 2001)

- Satisfying customer by delivering useful software early, frequently, and continuously
- Welcoming changing requirements even late in the development
- Co-operating daily with the business people throughout the project
- Considering working software as the primary measure of progress
- Relying on face-to-face conversation instead of intensive documentation as a communication tool
- Striving for simplicity in everything

In XP the development process is divided into release cycles and iterations. Release cycles typically last 1–3 months and they aim at delivering working software with added functionality to the customer. In the beginning of each release cycle is an exploration phase, where the customer's requirements that are to be included in the next release are chosen. After that, release cycles consist of iterations that typically last 1–3 weeks each. In each iteration developers implement a set of requirements. If all selected requirements cannot be implemented, the iteration's length is not increased, but some requirements are postponed into further iterations. (Beck and Andres, 2004)

The core values of XP are communication, simplicity, feedback, courage, and respect. These are pursued by a set of practices of which most had already been used independently in software development before XP was created. The ingenuity of XP was to choose practices with care so that they complement each other in the best possible way to achieve the goals of agile software development. These practices are presented in Figure 2 and explained in the following. (Beck and Andres, 2004)

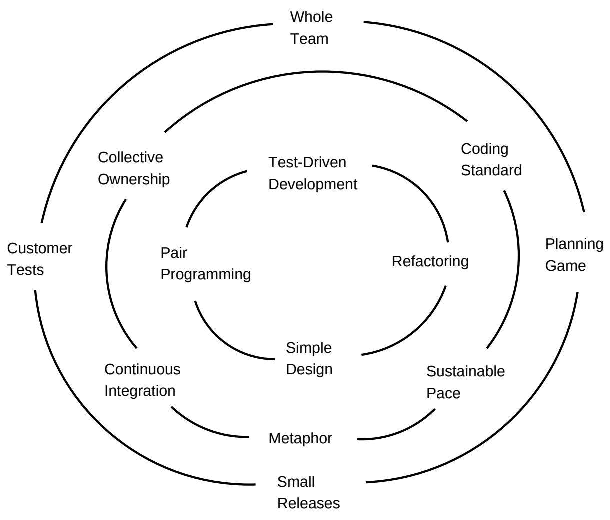

Fig. 2. XP practices. The outermost circle contains release cycle practices, the middlemost circle iteration practices, and the innermost circle daily practices.

- Whole Team
  The customer, which in XP means the one who uses the system not necessarily the one who pays the bill, should always be available for answering questions.
- Planning Game
  Planning game consists of release planning and iteration planning. Release planning occurs when the development of a new release begins. In it the customer's requirements are written on user story cards and the ones that are to be included in the next release are chosen. Iteration planning occurs in the beginning of each iteration. In it developers choose which user stories to implement in the next iteration and translate them into tasks that are then assigned to each developer.
- Small Releases
  Working software is released to the customer often with each release adding some functionality.
- Customer Tests
  The customer defines automated acceptance tests for each feature.
- Coding Standard
  A defined coding standard must be followed to ensure a consistent style for the source code.
- Sustainable Pace
  To ensure welfare no-one should work more than 40 hours a week.
- Metaphor
  A naming concept should be chosen so that it is easy to guess the functionality of a certain class or method from its name only.
- Continuous Integration
  Changes should be uploaded to the repository every few hours or whenever a significant break occurs in order to prevent integration problems further in the project.
- Collective Ownership
  Anyone is allowed to change any part of the code therefore i.e. enabling bugs to be quickly fixed. Pair programming supports this by sharing the understanding of the code and automated unit tests by detecting possible new bugs that the changes may cause.
- Test-Driven Development
  Automated unit tests are written for each non-trivial peace of code before the actual code is programmed.
- Refactoring
  The code must be refactored when it starts to become unnecessarily complicated. Possible symptoms for this are for example that changes in one part of the code start to have an effect on many other parts of the code.
- Simple Design
  The simplest way to introduce a certain functionality should always be chosen.
- Pair Programming
  All code is produced by two people programming on one computer. One does the actual coding and the other one reviews the code and thinks about the big picture. Roles should be traded and pairs mixed regularly.

4. User-Centered Design and Extreme Programming

4.1. Conflicting Elements of User-Centered Design and Extreme Programming

One of the main principles of XP is that working software is the primary measure of progress. This is a common goal with XP and UCD since in order for the software to be usable it has to be working. However, working software does not equal usable software. In XP working software emphasises the quality of the code, which may in fact decrease the developers' motivation to focus on the usability issues. (Bankston, 2003)

As discussed in chapter 3, XP works best with skilled individuals, who each take responsibility for their area of expertise. This should support the inclusion of usability experts in XP teams, but in practice these skills are often overlooked. Lack of usability experts makes it substantially more difficult to achieve good usability. (Seffah et al., 2005)

XP uses the term customer for the tasks that actually the users should do (Bankston, 2003). This may be misleading. For example, user stories are often written by the customer's representative, who probably does not fully understand the users' true needs (Constantine, 2001). This may lead to a product that has lots of features that the users do not really have use for, and that lacks features that the users would actually need, which may force the users to change their ways of doing things in order to be able to use the software (Patton, 2002). Relying only on the customer to provide the requirements may also lead to unclear and ambiguous user story cards, since the customer may not be capable of articulating, visualizing, or organizing the requirements properly (Bankston, 2003). Even if actual users are included in the team, there is a question of whether they represent typical users. It is all too often that the reasons for selecting the users are for example how popular or unpopular they are among their colleagues or that they just happen to be available. Even if the users represent typical users in the beginning of the project, they are bound to drift further away as they co-operate with the developers. (Hudson, 2003)

In XP user stories are used to capture requirements. However, as such user stories do not fit into expressing usability requirements. Thereby, usability requirements are typically dismissed in XP projects. (Jokela and Abrahamsson, 2004).

In UCD the user interface is designed and evaluated thoroughly before it is actually implemented. However, in XP the overall structure of the product (including its user interface) is not designed before the implementation is started (McInerney and Maurer, 2005). Instead XP focuses on implementing only a little piece of functionality in each iteration. Continuous integration, refactoring and unit testing guarantee that this does not lead to inconsistent code, but it is not so easy to guarantee that the user interface does not become inconsistent. In fact, it is almost certain that the general look&feel including navigation will require constant refactoring throughout the development process to maintain its consistency. (Bankston, 2003) This will be troublesome for users that have already begun to use the software (Constantine, 2001). I have also personal experiences on this. I was once asked to implement user interface screens for a mobile phone application. I was first given the requirements for one screen. After that was completed, another screen was requested, and finally after that, a third screen. Already by that time, it was very hard to maintain consistency, because the screens had so different functionality. Had I been given the requirements for all screens at once, I could have designed a general consistent look&feel before starting to implement the first screen.

One can see from ISO 13407 and Nielsen's model in chapter 2.3 that an essential component of UCD is usability evaluation. Although testing is emphasized in XP, it contains no practices that directly support usability testing (Bankston, 2003). There is no time to do thorough usability tests with users between iterations or release cycles and only testing paper prototypes and doing expert analyses do not provide an accurate picture of the product's usability (Constantine, 2001). However, releasing useful software early, frequently, and continuously to the customer may serve as ongoing usability tests (Armitage, 2004).

In general, XP does not pay attention to the usability issues and it may be that in its original form it works best in applications that are not heavily GUI intensive (Armitage, 2004). The belief that close collaboration with the customer guarantees good usability, is not valid. Instead the usability depends heavily on the abilities of the customer and the developers, since XP as a process does not support usability systematically. (Jokela and Abrahamsson, 2004)

4.2. Suggested Ways for Combining User-Centered Design and Extreme Programming

To make sure that developers are capable and motivated in taking usability issues into account, it is suggested that user interface designers should be partnered with the developers in XP pair programming practice (Bankston, 2003). If this is not feasible, then at least the user interface designers should always be available to answer the developers' questions and to validate their work (McInerney and Maurer, 2005).

To make sure that the users' needs are truly considered, it should be emphasized that XP's term customer includes the actual users (Bankston, 2003). The context of use (who are the users, which tasks they perform, what are the physical and social surroundings, etc.) should be analysed in the beginning of the project (Hudson, 2003). Preferably the users and/or domain experts should be included in the development team to answer the developers' questions and to give them feedback on their work (McInerney and Maurer, 2005). User interface designers should also make sure that the customer is able to properly articulate the requirements before the actual user story cards are created (Bankston, 2003). Also, the support for usability requirements should be included in the user story cards (Jokela and Abrahamsson, 2004).

There are some suggestions to ensure that the incremental implementation without designing the overall structure of the product beforehand does not break the user interface. To begin with, the overall layout of all user interface elements should be sketched, a versatile navigation scheme designed, and a look&feel style guide defined, before the actual implementation is started (Constantine, 2001; Jokela and Abrahamsson, 2004). Also, during the iterations the user interface design should always stay a few steps ahead of the actual implementation. In each iteration user interface prototypes for upcoming iterations should be designed and an unofficial visionary prototype for the final product maintained. It would also be a good practice to maintain a design rationale that one could use to check the reasoning behind user interface design decisions later on. (McInerney and Maurer, 2005)

Usability evaluation with users should be included as part of the acceptance testing process (Bankston, 2003). Also, enough time should be allocated for improving the user interface based on the evaluation results in the beginning of the next release cycle (McInerney and Maurer, 2005).

Since XP does not take usability into account and UCD practices as such are too heavy to be included in XP, some lightweight UCD practices have been suggested. Most notable is the process suggested by Constantine (2001). It emphasizes the modelling of user roles and tasks in order to create complete task scenarios and user interface prototypes. The process should be integrated into XP's planning game practice and executed in the beginning of each release cycle. The process should be completed before the implementation of the components related to user interfaces is begun. However, system's internal components may be implemented in parallel. The process does not require heavy documentation since the possibly vague paper prototypes can be explained to the developers following the XP's principle of face-to-face conversation being the most important communication mechanism. The different phases of the process are presented in the following. (Constantine, 2001)

1) Stakeholders (domain experts, business people, developers, users, etc.) are recognized and gathered together.
2) The current situation is explained by domain experts and users.
3) Expected functions, concerns, etc. are brainstormed.
4) User roles are brainstormed on index cards.
5) User roles are prioritized based on how relevant they are for project's success.
6) Task descriptions of different user roles are brainstormed

on index cards.

7) Tasks are prioritized based on how common they are and categorized into three classes: required, desired and deferred.
8) Required tasks and desired tasks that are interesting are described in more detail.
9) Tasks are organized into groups based on how likely they relate to each other. Index cards are duplicated if necessary.
10) A paper prototype is created, where each screen should represent one task group. The focus is on required tasks, but all tasks are kept in mind for the user interface's consistency.
11) Paper prototype is tested with users based on scenarios derived from the task descriptions.

Patton (2002) describes the following experiences on using the process described above. The brainstorming session in the beginning provided good background information and its results could be used later to check whether all issues had been considered. Deriving paper prototypes from task descriptions was an easy and efficient way of transferring the knowledge of what needs to be done into what it could look on the screen. Paper prototypes were created using post-it notes, which were easy to move around. Also, the process helped in creating a common understanding between the developers and the customer. In addition to this, the task descriptions helped the testing process later on. On the negative side of things, the process proved to be quite heavy and exhausting to be a part of lightweight XP. This was solved by creating the paper prototypes on a later time with a smaller group of people. Also, the paper prototypes were not accurate enough for the developers to base their implementations on and the designers weren't always available to explain things face-to-face. Therefore, the final paper prototypes were cleaned up and the user roles and task descriptions were briefly documented for the developers. (Patton, 2002)

5. Conclusions

This study focused on identifying conflicts between UCD and XP as well as discussing ways for resolving those conflicts. The main findings are summarised in table I.

TABLE I
CONFLICTS BETWEEN XP AND UCD AND SUGGESTIONS FOR RESOLVING THEM

XP focuses on	**UCD focuses on**	**Conflict**	**Suggestions for resolving**
Quality of the code	Usability of the end product	XP developers are not motivated to consider usability issues	UI designers always available to help the developers, usability evaluation with users integrated into acceptance testing
Satisfying the customer	Concentrating on the user	If users' true needs are not identified, the end product does not satisfy the users in the best possible way	Users included in the development team, lightweight process for interpreting the users' needs and translating them into paper prototypes
Designing only what is necessary at the time	Proper design upfront	User interface becomes inconsistent if it is not designed properly	Overall layout, navigation and look&feel designed upfront, UI prototypes designed for upcoming iterations and a visionary prototype of the final product maintained throughout the process

The main thing about the suggestions is that they cannot break the lightweight nature of the XP process. Constantine (2001) presented a lightweight method that covered three of the four UCD activities defined in ISO 13407. Covered activities were understanding and specifying the context of use, specifying the user and organisational requirements, and producing design solutions. The activity that was not covered was evaluating designs against requirements. Although it was suggested that usability evaluation with users should be included in acceptance testing (Bankston, 2003), and that sufficient time should be reserved to the beginning of the next release cycle for correcting the problems found in the usability evaluation (McInerney and Maurer, 2005), no practical ways for achieving these in a lightweight manner were given. This could be an interesting concern for further study.

The study consisted of only three articles that reported actual experiences on combining UCD and XP. Other articles presented more inferred thoughts on how UCD and XP should be combined. Of course, these thoughts were drawn from actual experiences. In science, it is often the case, that reports are written only on succesful projects. However, in this analysis one of the articles (McInerney and Maurer, 2005) reported mainly positive experiences, another one (Patton, 2002) positive and negative experiences, and the third one (Jokela and Abrahamsson, 2004) mainly negative experiences. Nevertheless, more analysed cases would have given more grounds to base one's conclusions on, so the results presented in this paper should be seen only indicative of the ways how UCD and XP can be combined.

For further discussion, see the Yahoo discussion group ”Agile Usability”, which aims at connecting the usability community to the agile development community (Patton, 2006).

References

Armitage, J. (2004). Are agile methods good for design? *interactions 11*(1), 14–23.

Bankston, A. (2003). Usability And User Interface Design In XP, White Paper. `http://www.ccpace.com/Resources/documents/UsabilityinXP.pdf`.

Bass, L. and B. E. John (2001). Supporting usability through software architecture. *Computer 34*(10), 113–115.

Beck, K. and C. Andres (2004). *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional.

Beedle, M., A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, K. Schwaber, J. Sutherland, and D. Thomas (2001). Manifesto for Agile Software Development. `http://agilemanifesto.org/`.

Bosch, J. and N. Juristo (2003). Designing software architectures for usability. In *Proceedings of the 25th International Conference on Software Engineering*, pp. 757–758. IEEE Computer Society.

Constantine, L. L. (2001). Process agility and software usability: Toward lightweight usage-centered design. *Software Development 9*(6).

Hudson, W. (2003). Adopting user-centered design within an agile process: A conversation. `http://www.syntagm.co.uk/design/articles/ucd-xp03.pdf`.

ISO (1988). Guidance on Usability. Technical report, ISO 9241-11.

ISO (1999). Human-centered design processes for interactive systems. Technical report, ISO 13407.

ISO (2000). Software Engineering - Product quality - Part 1: Quality model. Technical report, ISO 9126-1.

Jokela, T. and P. Abrahamsson (2004). Usability assessment of an extreme programming project: Close co-operation with the customer does not equal to good usability. In *5th International Conference, PROFES 2004, Kansai Science City, Japan*, pp. 393–407. Springer Berlin / Heidelberg.

McInerney, P. and F. Maurer (2005). Ucd in agile projects: dream team or odd couple? *interactions 12*(6), 19–23.

Nielsen, J. (1993). *Usability Engineering*. Boston, MA, USA: Academic Press.

Patton, J. (2002). Hitting the target: adding interaction design to agile software development. In *OOPSLA '02: OOPSLA 2002 Practitioners Reports*, pp. 1–7. ACM Press.

Patton, J. (2006). Agile Usability Yahoo Discussion Group. `http://groups.yahoo.com/group/agile-usability`.

Seffah, A., J. Gulliksen, and M. C. Desmarais (2005). *Human-Centered Software Engineering - Integrating Usability in the Software Development Lifecycle*. Springer.

Seffah, A. and E. Metzker (2004). The obstacles and myths of usability and software engineering. *Communications of the ACM 47*(12), 71–76.