Combining Extreme Programming & Interaction Design

Motivation

I am interested in the question of how can **Extreme Programming (XP)** and **Interaction Design** be combined or if it is possible at all.

I learned about **Extreme Programming** for the first time during a course held by Prof. Robert Duisberg at the University of Technology in Vienna last year. Prof. Duisberg taught the principles of this new software development methodology and we practiced them by doing a class project. I experienced this methodology as a completely new way of doing software development which is, in many concerns, superior to the traditional waterfall model. Since then I became very interested in different issues regarding XP, i.e. if it can be applied to other non-traditional software engineering tasks like web development.

Interaction Design deals with user related issues and tries to build products starting from the users point of view and not from the implementation or technological side. This "user-centered design" approach is proposed by *Alan Cooper*, a very well known author in the usability community. He positions the task of interaction design as the first phase prior to the other phases of the waterfall model. It is important for him that no line is coded before interaction design is done in order not to influence or narrow interaction issues by pieces of the product that are already present.

From my point of view both mentioned methods are important improvements to the development process mostly performed today. But since extreme programming is a rapid prototyping approach where design is done all along the process it is contradictory to the method of interaction design that should be done before implementation starts. Therefore I would like to investigate if there is a way to combine both worlds and build up a methodology that improves both, the usability of a product and the development process itself.

Introduction to Extreme Programming

The **Extreme Programming** software development methodology was conceived 6 years ago by *Kent Beck*. It presents an alternative to the nowadays mostly used waterfall model. *Kent Beck* took a careful look at all shortcomings of the waterfall model especially from a practical rather than a theoretical point of view and came up with a radically new and controversially discussed idea of extreme programming.

Some Problems and shortcomings of using the waterfall model:

- Requirements change all along the process
  The waterfall model is based on the theory that all requirements and characteristics of a system can be provided in the early phases – analysis and design. But in most cases this isn't true. Requirements and user needs change and/or become more clear all along the process. In order to that, all analysis and design documents have to be updated continuously.

- Errors detected late are very expensive
  Testing is the very last phase within the waterfall development process. If errors and problems are detected in this stage that have their cause in earlier phases, correction is very expensive. You have to go back to the design phase, alter all documents, correct the implementation and test again.

- Heavily dependent on individuals
  Since every person in the development team is responsible for a certain part of the system, they "own" these parts. If a member of the team leaves, big problems arise because no one else is as familiar with the code.

- Most projects are late
  Because it is very hard to estimate months or even years in advance, most of the estimations are incorrect and fall short. Another fact that leads to the fact that most projects are late is, that the waterfall model is not built for a constantly changing environment and requirements. If a problem is detected at some stage, all previous ones may have to be altered, which is very time consuming.

- Most projects are feature driven
  Most customers don't really know what they want and therefore the wish list of features is very long. ("This and that could maybe be useful...") Because of the construction of the development process, every feature for the end product has to be considered from the very beginning. And since the customer doesn't get feedback except of abstract documents and rough prototypes, she has to wait until the end of the project to see the real product that maybe contains many features that aren't really needed.

- The customer feedback loop is very long
  As mentioned before the customer gets a feeling for the product just at a very late stage of the development process. If she discovers that this isn't what she imagined and wants changes, you have to go back and fix everything from the very beginning which is very expensive and time consuming.

What is so extreme about XP?

Kent Beck took a look at all the good practices in software development and pushed them to the extreme, therefore the name: "Extreme Programming".

- If short iterations are good – make them really, really short.
  - Small Releases
  - Two week iterations
- If code reviews are good – review code all the time.
  - Pair programming.
- If testing is good – everybody will test all the time.
  - Integrated unit testing and acceptance testing.
- If design is good – design is an ongoing daily business.
  - Constant refactoring.
- If simplicity is good – always choose the simplest thing.
  - The “simplest thing that could possibly work.”
- If integration testing is important – integrate and test several times a day

[See lecture slides of Prof. Duisberg]

Extreme Programming practices and principles:

- The “Planning Game”
  - Formalized Communication, Negotiation (Requirements)
  - Feature/Task Specification, Estimation, Prioritization
  - Formal use of note cards for “Stories” and tasks.

- Continual Testing
  - Tests are written before functionality is implemented.
  - The Test Utility is the console. Run constantly.
  - Tests are never thrown away.

- Small Releases
  - An extremely simple version in production **immediately**
  - Automatic build system runs the full test suite.
  - Every integration (many times a day) generates a release.
  - Two week iterations.

- The Simplest Possible Design at any moment
  - Do **not** design for the future! (This goes against the grain)
  - Extra complexity & unused code is removed as soon as it is discovered

- Pair Programming
  - All production code is written with two programmers at one machine.
  - USB keyboards & mouses, 2 at each workstation.

- Collective Ownership
  - Anyone can change any code at any time.

- Continuous Integration
  - Task granularity = ~ 3 hours' work => test & integration.

- Continual Refactoring
  - Restructure system to simplify, or improve, without changing function.
  - Constant process of design refinement.

- On-site Customer
  - Available full time for questions/design.

- 40 Hour Week
  - Overtime is strongly discouraged.
  - Overtime two weeks in a row is prohibited.

[See lecture slides of Prof. Duisberg]

These principles may seem unusual at first sight. Above all it requires a shift in the way of how programmers think and away from many practices they are trained and experienced in. It is also not possible to implement only a part of these principles. Either you do all of them or you don't do XP at all. This doesn't imply that these rules are strict and unchangeable. But all of them have to be present in a certain way in order to perform extreme programming.

The User in the XP Development Methodology

In this chapter I want to discuss the role of the user in the Extreme Programming development methodology, show how the user is involved, point out the differences to the traditional waterfall model and also talk about the shortcomings and problems of that kind of involvement.

First of all, I want to introduce the key principle in XP for that matter: the **On-Site Customer**. The on-site customer is an employee of the client who is working together with the XP development team. As the name "on-site customer" implies, this person works physically at the same workplace as the developers do.

The tasks of the on-site customer are:

- describing and clarifying the user stories to minimize misunderstanding between developers and the customer
- act as available resource for the developers in case of client related questions
- writing tests for the user stories
- testing the releases

The benefits of having a client easily available are obvious: Whenever developers are not certain about how a piece of the product is intended to work, what the key issues of a problem out of the customer's view are or any other client related question, they can easily be discussed right away without any delay, calling the client, scheduling meetings and so on. Because of that fact developers don't tend to make assumptions about certain unclear issues but rather contact the on-site customer. Therefore false assumptions, development of unnecessary functions or working in a wrong direction are minimized.

These advantages are not only valid for the functional view of the product but also for usability and interaction issues. The on-site customer is in charge of controlling and enhancing the usability and design of the interaction patterns constantly throughout the whole development cycle.

A concern related to the on-site customer is, that she maybe gets too involved in the work of the development team and tends to see various aspects through the eyes of a developer rather than a user, thus becoming blind for user related issues. This concern is also raised in the XP methodology and it is recommended to replace the on-site customer after a certain amount of time.

Another problem is, that the on-site customer has to write tests for the user-stories. Despite the fact that those tests don't necessarily have to be tests in a programmatic sense but can rather be textual descriptions of the intended functionality and interaction patterns, the on-site customer will most of the time be a technology and computer savvy person. But that means also that this person is mostly not a good representative of the average user of the product. The singular role name "on-site customer" implies furthermore that only one person represents that role. This is also underlined by the fact that this person is provided by the client, who is "losing" her employee temporarily. But that means again that the user's issues are only represented by one person thus emphasizing her point of view.

The waterfall model on the other side involves the user only at certain points within the development process. First of all of course in the first phase, analysis. In this phase, the analysts work together with the client on a precise problem and project description resulting in a document called "requirement specification". This analysis contains both, functional and usability/interaction views. Ideally the analyst is visiting a number of different users at their workplace, watching and analyzing how the tasks that should be modeled in the product are actually performed and thus helping to provide a natural mapping between the real world and the conceptual model of the product. This process is also known as "ethnography"[3], but only applied in very few, mostly academic projects because of its time- and resource consuming nature.

In the second phase of the waterfall model, the design phase, the developers are working on a plan on how to actually implement all the intended tasks and goals. During this process, the user is usually not involved and only contacted in case of major unclarities or problems. The end of this phase is marked by a document called "functional specification", which includes the specification for the implementation of the product in various aspects as well as design prototypes. At this point, the user comes into the game again: her task is to investigate and test the prototypes and provide feedback for improving the design.

After that, in most cases the longest phase, implementation, begins. Assumptions are made by developers in case of not clearly specified aspects because asking the customer herself is most of the time related with big bureaucratic efforts if they can't be resolved in regular project meetings.

In most projects only at the very end of the implementation or even test phase, the user is involved again. But at this point almost every aspect of the product is set and major changes can hardly ever be made.

Interaction Design Basics

Alan Cooper's method of **interaction design** is an attempt to improve the waterfall model by introducing a phase of "interaction design" at the beginning of the development process where interaction and usability related issues are carefully investigated and designed by specialists. In this case not the functional and feature based aspects but rather user based aspects are emphasized. "User interface design" is only a small part of "interaction design", whereas interaction design is placed on a higher level and effects the product at an earlier and more fundamental stage.

The main techniques of interaction design are **Personas**, their **goals** and **scenarios**. Whereas **personas** are a created cast of characters representing real persons along with both their knowledge in the computer and the application domain. These personas have certain **goals** they want to achieve when using a product. A **scenario** is basically a detailed story about a person performing a certain task to achieve her goals. Alan Cooper states furthermore that designing for only a few main characters and their goals helps to enhance the usability of a product and avoids feature driven dancing bearware.

Interaction Design is basically an add-on to the waterfall model that enhances it but still carries many of the disadvantages.

Combining both worlds

*In the following chapter I am trying to point out the frictions between **Interaction Design** and **Extreme Programming**, discussing possible combinations and presenting a solution which is most promising.*

At first I want to start with discussing an interview between *Kent Beck*, the father of Extreme Programming, and *Alan Cooper*, the father of Interaction Design. This article appeared at the beginning of this year in the web journal "Fawcette Technical Publications"[4] and is titled "Extreme Programming vs. Interaction Design – When two development design visionaries meet, there's room for consensus – but not much".

Alan Cooper says *"It's my experience that neither users nor customers can articulate what it is they want, nor can they evaluate it when they see it."*[4] and also points out that the communication structure used in Extreme Programming is better than traditional ones but still can't overcome this fact and the related problems. After all I think that this statement might be mostly true but stated far too drastically here. I think that customers can tell very well what they want, maybe only at a higher level, and even more they can tell what they don't want if they see it.

Furthermore Beck and Cooper argue about if Interaction Design has to be done before the construction starts and be a phase rather than a part of a network like structure used in XP. One argument of Alan Cooper in favor of having Interaction Design before construction begins is: *"It has to happen first because programming is so hellishly expensive..."*[4]

Following this they discuss the nature of software engineering:
Cooper: *"Building software ..., it's more like building a 50-story office building or a giant dam."*[4]
Beck: *"If you build a skyscraper 50 stories high, you can't decide at that point, ok, we need another 50*

stories ..."[4]

Here, Kent Beck is pointing out a very important issue: that software engineering is not like building houses, the requirements change frequently and often essentially. There are no general standards or laws on how to build software, not even for just parts of software products.

This discussion lead to a further investigation of the nature of Extreme Programming and Interaction Design: Extreme Programming is a method to cope with the fact of requirement changes and allows quick responses to them. Interaction Design on the other hand tries to dampen this frequent changes by being a layer in between the client and the developers and providing a higher level view.

Related to the combination of Extreme Programming and Interaction Design, Kent Beck says: *"To me, the shining city on the hill is to create a process that uses XP engineering and the story writing out of interaction design. This could create something that's really far more effective than either of those things in isolation."*[4] On the contrary, Alan Cooper sees the combination of Extreme Programming and Interaction design in a way that Interaction Design is done by a team of specialists together with the client at the beginning of the project and then the provided product proposal is implemented by the XP team.

Out of this interview, we can deduct two possible combinations of Extreme Programming and Interaction Design:

1. As prephase before the XP process begins
2. Integrated in the XP development process

The problem of the first one of these combinations is primarily that it goes against the principles of Extreme Programming. The basic assumption of XP is that you can never cover every aspect and requirement of a product at the beginning, in an analysis or design phase. "Important things first" and "requirements will change throughout the development" are basic rules of Extreme Programming.

The problem of the second combination is that Interaction Design is maybe tied too much to the technical and internal issues of the product construction. Interaction Design is intended to start from the user's side and not to deal with implementation issues. Interaction Design is situated at a higher level than the XP development process.

Now what? Impossible to combine?

Extreme Programming and Interaction Design are two new attempts trying to improve the way software is produced nowadays. But basically they are not addressing the same issues.

Interaction Design's main task is to improve the usability and behavior of a product, not the manufacturing process itself, but of course influencing it.

Extreme Programming's main focus on the contrary, lies at the manufacturing process itself. It tries to solve problems, software producers are suffering from and creating a win-win situation for customers and developers. Extreme Programming also includes behavioral design, mainly via story-writing, but it's not a major intention of XP.

What I propose is a combination of Extreme Programming and Interaction Design at two different levels. It can be considered as a compromise of the two above mentioned combinations. This process would look as follows: The XP development team has a pair of members who are interaction design specialists. Those two members are doing the very first iteration in combination with the customer to draw the behavioral outline of the product, finding *personas* and defining the general *goals*. Due to the fact that goals and personas are situated at a higher level, it is very unlikely that they change much throughout the process. This first iteration shouldn't cover any details but give a rough

sketch of the product as basis for the developers as well as the client.

After this very first iteration, the whole XP team starts to roll in full motion. Now the interaction design team is in charge of writing the user stories together with the customer and designing interaction and behavior of the product at a finer granularity for each iteration. By applying another basic method of Extreme Programming, refactoring, usability and all user-related issues are improved over time. Because of the fact that XP developers don't hesitate to refactor major parts of the product throughout an iteration, the danger of designing yourself into a corner is minimized. This is also true for the behavior of software. It is very unlikely that the first design is perfect and therefore needs several steps of improvement. Extreme Programming provides this mechanism inherently through the rapid development cycles.

I think this combination is getting most out of both concepts by combining their advantages and minimizing their shortcomings. This new methodology can't be applied to every kind of software project, but it is a major improvement for many of them by increasing both, customer and developer satisfaction.

References

1. Kent Beck, **Extreme Programming Explained**, Addison-Wesley, 2000.
2. **Extreme Programming Website**, http://www.extremeprogramming.org/
3. Simonsen, J. Kensing, F., **Using ethnography in contextual design,** Communications of the ACM, 40(7): p 82-88, 1997
4. Elden Nelson, **Extreme Programming vs. Interaction Design**, A Interview with Kent Beck and Alan Cooper, Fawcette Technical Publications, January 15, 2002, http://www.fawcette.com/interviews/beck_cooper/
5. Alan Cooper, **The Inmates Are Running The Asylum**, SAMS, 1999
6. Donald A. Norman, **The Invisible Computer**, MIT Press, 1999
7. Jakob Nielsen, **Usability Engineering**, Morgan Kaufmann Publishers, 1994
8. David Astels, Granville Miller, Miroslav Novak, **A Practical Guide to eXtreme Programming**, Prentice Hall PTR, 2002
9. **Extreme Programming Website,** http://www.xprogramming.com