Software Engineering, eXtreme Programming and a WAP-browser

Afstudeerscriptie Informatica
Faculteit der Natuurwetenschappen, Wiskunde en Informatica (FNWI)
Universiteit van Amsterdam
Afstudeerrichting: Programmatuur
Afstudeerdocent: prof.dr. P. Klint
Datum: Januari 2002

Wouter van der Kamp

Abstract

XP, or eXtreme Programming, is a methodology for teams developing software in the face of vague and rapidly changing *requirements*. I did research into this subject by using it while developing a WAP-browser; the results are presented in this paper.
The features of XP include pair programming, working with iterations and continuous integration.
Pair programming means that all programming is done with two programmers working on one computer. Programmers who work in pairs are less likely to make mistakes.
The lifecycle of every XP project is divided into *iterations*. With every iteration the architectural design is adjusted along the needs of the iteration; this is very different from the common software engineering practice to make the whole architectural design at the beginning of the project.
Continuous integration means that every *task* is integrated after its completion. The integration is immediately followed by the running of a program that tests every bit of code in the program. By doing this the programmers make sure that the integration did not break any working functionalities.
The apprenticeship consisted of four complete iterations. Due to practical reasons, some of the practices of XP could not be tested. I decided to extend my research by studying some software engineering literature.
At the end of the apprenticeship I was able to make some comparisons between XP and the literature, give some suggestions about improving XP and conclude that XP has some practices that are valuable in certain circumstances.

Table of Contents

Introduction

In this chapter I introduce the company Catchy and the technologies that were used during the apprenticeship. Also my motives and goals for this project can be found here.

XP

XP is a methodology for teams developing software in the face of vague and rapidly changing *requirements*. XP is an abbreviation for 'eXtreme Programming'. 'Extreme' refers to the fact that with XP some software developing practices are put to extremes, for example:

- Code reviews of *all* code
- Testing of the whole program with every integration
- Iterations as short as possible

XP is discussed in more detail in chapter 1. More on eXtreme Programming can be found on http://www.ExtremeProgramming.org/.

WAP

WAP is the abbreviation of 'Wireless Application Protocol'. It's a specification that can be used to develop wireless applications. The specification defines, amongst others, WML, the 'Wireless Markup Language'. WML is, like HTML, a language that is used to mark-up text and also makes user-interaction possible. All the WAP specifications can be found on http://www.wapforum.org/.

WML

WML is the language in which WAP-pages are written. It is designed to display content on a wireless device, so it takes into account that the display is probably small, there are limited user input facilities, small memory, etc. One WML file contains one or more cards, the deck. The WAP-browser usually displays one card at a time. At the time the project started version 1.2 of WML had just been released by the wapforum. So the team decided to make the WAP-browser conform to WML 1.2. Its DTD (Document Type Definition) can be found at http://www.wapforum.org/DTD/wml12.dtd.

Catchy

Catchy is a company that develops applications that use the mobile internet. The company was established in the year 2000, after some ideas for products were developed. The WAP-browser was one of these ideas; the program was going to be called 'Mobilizer'. Founder and president of Catchy is Ruben Brave. More info on http://www.catchy.net/.

Motivation

The job of building a WAP-browser for Catchy was something that came on my path, and thereupon I decided to make this job an apprenticeship. So the motivation for this apprenticeship is mainly a practical one. I took this job because it's satisfying for me to be able to build something for the future; it has the advantage of less (time-) pressure, and makes you feel like a pioneer.

Objective

The objective of this apprenticeship was to research into XP by using it while developing a WAP-browser. I will try to determine the usability of XP in the software-developing world, and share our experiences with the world.

1. eXtreme Programming

In this chapter the ideas from the book 'Extreme Programming' by Kent Beck ([XP]) are described briefly.

XP is a methodology for teams developing software in the face of vague and rapidly changing *requirements*. XP is an abbreviation for 'eXtreme Programming'. It tries to be a solution for many problems that arise during software development. A few examples of such problems:

- The product is not finished in time
- The product does not work properly, and consequently is not used
- The product does not do what it was intended to do
- When the product is finished, it's no longer interesting from a business side of view

The theory behind XP states that these problems are the result of a lack of the following four values:

1. Communication
2. Simplicity
3. Feedback
4. Courage

All practices of XP are based on using these values.

The Practices of XP

Planning
[XP] describes a game that must be played at the beginning of every *iteration*, it is played by business and development people. The objective of the game is to come up with a good planning for the iteration. First the business people write a text that states what the final product must be able to do. Then they decide what the most important bit of software in the product is, this bit is built in the first iteration. The business people must write functional tests that must work after the iteration; the development people estimate how long the iteration will take. It's also estimated how much iterations the whole product will take. When the game is over, the business and development people have decided exactly what will be done in the coming iteration and what the deadline of it is.
Within the iteration the work is divided into *tasks*. Programmers must successively estimate the amount of time the task is going to take, implement the task and verify that the task does what it was supposed to do.
After the iteration the development and business people come together again; the business people can now see what has been build. When the iteration took longer than planned, it's clear that the whole project will take more time than planned. Notice that the business people know this after just one iteration. It could also happen that the business people are not completely satisfied by something that was build, because of some misunderstanding. The advantage of working with iterations is that these kind of things are noticed relatively early.
A new iteration is started, so the planning game starts again. It is decided what the most important bit of software left to build is, etc.

After a few iterations, the planned deadlines will get more and more realistic.
Working with iterations and discussing the product with business people within iterations results in more communication and feedback in comparison with other planning strategies. Communication and feedback are two of the four values of XP.
(More about planning in chapter 7.)

Small Releases
Every *release* must be as small as possible. This means that the software must have only the most important features. It is also important that the release cycle is as short as possible, yet it may not contain features that don't work completely.
Feedback and simplicity (two of the four values) are used in this practice. A small release keeps things simple, short release cycles result in more and faster feedback.

Simple Design
The general thing you here about design is 'Implement for today, design for tomorrow'. But the problem with this motto is that the future is often uncertain. Therefore [XP] states that it is better to have a design that is as simple as possible, instead of a design that reckons with things that will never see the light of day.
According to [XP], the right design is the one that

- Runs all the test (that are written beforehand)
- Has no duplicate logic
- States every intention important to the programmers
- Has the fewest possible classes and methods

The design is done just before the implementation starts, and is only done for the coming iteration. Of course it could happen that in an iteration the architecture of a previous iteration must be altered. This problem is partly solved by the fact that an XP team will quickly get experience in changing 'old' code, because an XP team does a lot a refactoring (see 'Refactoring' in this chapter).

Testing
In XP, the whole program is tested on a regular basis, viz. every time code is added to the program. Automated tests run through the whole program and are making sure that every bit of code still works. This means that with every bit of code that is added to the program, a test for it must also be written. These tests are saved and executed as long as that bit of code is present in the program. The programmers should always keep all the tests running, this way the programmer that has added code and finds a test that doesn't run knows that this is caused by his code.
(More about testing in chapter 8.)

Refactoring
Programmers must restructure the system, when the system can be simpler or more flexible. By working this way the programmers try to make adding features in the future as simple as possible. Maintainability is very important in software engineering and esspecially XP, as working with a design that only covers the current iteration provides the possibility that a lot of code needs to be changed at some time.

Refactoring brings a risk, it is possible that by refactoring some code, some functionality that is depending on that code 'suddenly' does not work anymore. XP solves this problem by using automated tests; the errors are discovered early and can quickly be repaired.

Pair Programming
All code is written with two programmers programming at the same terminal. One is sitting behind the keyboard, and the other one is watching and thinking ahead. So now a *code review* of all code takes place. Pairing is dynamic, so when two people pair in the morning, they might pair with other people in the afternoon. Programmers decide themselves with whom they pair.
(More about pair programming in chapter 6.)

Collective Ownership
Every member of the team can change every bit of code in the program, as long as they make sure that all tests keep running. By this XP tries to avoid the situation that the progress of the work slows down when some member of the team is not present, and by that his code in not accessible.

Continuous Integration
[XP] suggests that one machine must be dedicated to integration. When a *task* is finished, the programmers integrate their code into the system and run all the tests. When a test fails it is obvious that it is caused by their code, as it is forbidden to leave the integration-computer without all tests running.
(More about integration in chapter 8.)

Coding Standards
Programmers must code with the same *coding conventions*. This is important because the code is often refactored, the architecture is often changed and the programmers often work on different parts of the code and also with different partners. Without coding conventions the code would become a undecipherable.

2. Mobilizer, Introduction

In this chapter I describe Catchy's motivation for building a WAP-browser, it's motivation to use XP and other thoughts that Catchy had just before it started the Mobilizer project.

Catchy was established in the year 2000 as a company that builds applications that use the mobile internet. As WAP was the only mobile platform that was operative at the time, this meant that Catchy was going to build WAP-pages. WAP-pages are the equivalent of Web-pages and aren't written in HTML ([HTML]) but in WML, the Wireless Markup Language; WAP-pages can be viewed with WAP-browsers, the equivalent of Web-browsers. These WAP-browsers were and are developed for telephones, palms but also for desktop computers. This is handy, as developing a WAP-page with a mobile telephone can become very expensive. The state of the desktop WAP-browsers at that time was very poorly; every browser had some malfunction.

This all meant that Catchy had a problem: it didn't have a proper tool for building WAP-pages. So it decided to develop one on its own. Maybe it could sell it to developers when it was finished or, better yet, maybe it could port the browser to a phone, and make some money that way. The WAP-browser was going to be called 'Mobilizer'.

Catchy wanted to develop a WAP-browser that looked like a Web-browser, so the audience could easily be introduced to WAP, as it probably would already be familiar with Web-browsers. The WML language has many similarities with HTML; it, for instance, also uses anchors to link one WAP-page to another and enables the writer to use italic, bold and underlined fonts. A difference with HTML is that WML can only display one type of picture, viz. WBMP (Wirless Bitmap) pictures; WBMP pictures only have two colours. This is because the WAP-pages will (usually) be displayed on wireless devices, so the display would normally only have two colours. Another difference with HTML is that the WML specification doesn't allow tables to be located in other tables, which is allowed in HTML. This is also something that shows that the WML specification reckons with the fact that small devices must be able to display WAP-pages.

Catchy realized that this was going to be a big project, the WAP-browser would consist of a WML-parser, a WML renderer, a module to connect to a WAP-server, etcetera. As the developers had not much experience in a project of that size, and making a technical specification of a complete WAP-browser was thus thought to be a big problem, Catchy decided to use XP. With using XP, it thought, there is no need for writing a complete technical specification, as it is only necessary to write a technical specification for the coming iteration.

Sadly, just before the project started the team downsized from the initial planned four to just two programmers. It was clear that this would have negative consequences on my research, but I decided to carry on.

3. Mobilizer, Functional Specification

In this chapter I describe the WAP-browser we had in mind at the beginning of the project. This text is the starting point of the Planning Game. Normally business people would write this text.

Interface

The WAP-browser must have a user-interface as depicted in figure 3-1.

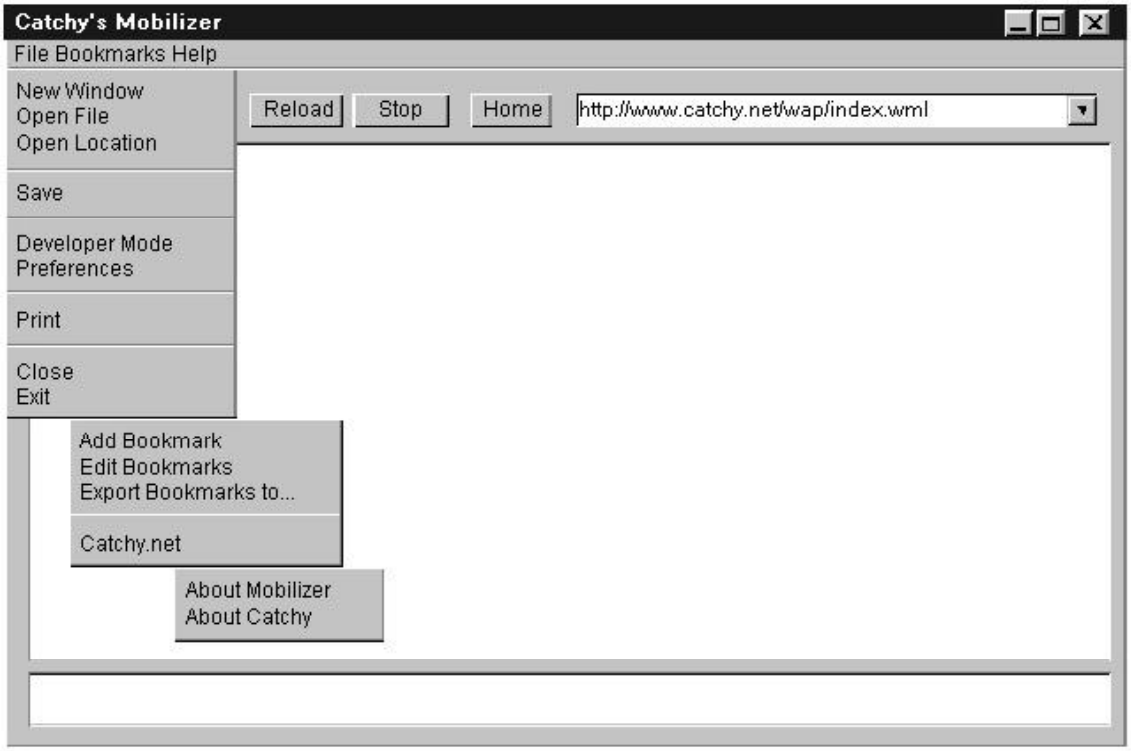

Figure 3-1, user-interface of the WAP-browser

The buttons that are covered by the menu are the 'Back' and 'Forward' buttons. The user can type an URL in the location bar that is located at the upper right of the window. When the 'Return' button is pressed the WAP site is displayed in the big sub-window, the content window. Located under the content window is an info window that displays error messages and status info. Pressing the various buttons or choosing the various menu items will have the following consequences:

Menu item / button	Consequences
New Window	A new window is opened so that the user can visit more than one WAP-site at a time
Open File	A pop-up window lets the user choose a local WML file, this file is successively shown in the content window
Open Location	A pop-up window lets the user type an URL. The program receives the WML file and displays the content in the content window
Save	Let's the user safe the current WML file to a local disk
Developer Mode	Gives the program the functionalities needed to develop WAP-pages, see below for details
Preferences	Let's the user choose his/her preferences

Print	Prints the current WAP-page
Close	Closes the current window, only available when the program has more than one window
Exit	Closes all windows and exits the program
Add Bookmark	Adds the current WAP-page to the bookmark list
Edit Bookmarks	Lets the user edit the bookmark list
Export Bookmarks to...	Lets the user export the bookmarks to a ftp location
About Mobilizer	Displays information on the Mobilizer program
About Catchy	Displays information on the company Catchy
Back	The previous WAP-page is shown
Forward	Let's the user go forward in the history list, only available when the current page has been reached by means of the back button
Reload	Reloads the current page
Stop	Stops the current download action, only available when the program is busy with downloading files
Home	Goes to the home location (that can be set in the preferences)

Developer Mode

When the user chooses developer mode the program changes, as figure 3-2 shows.

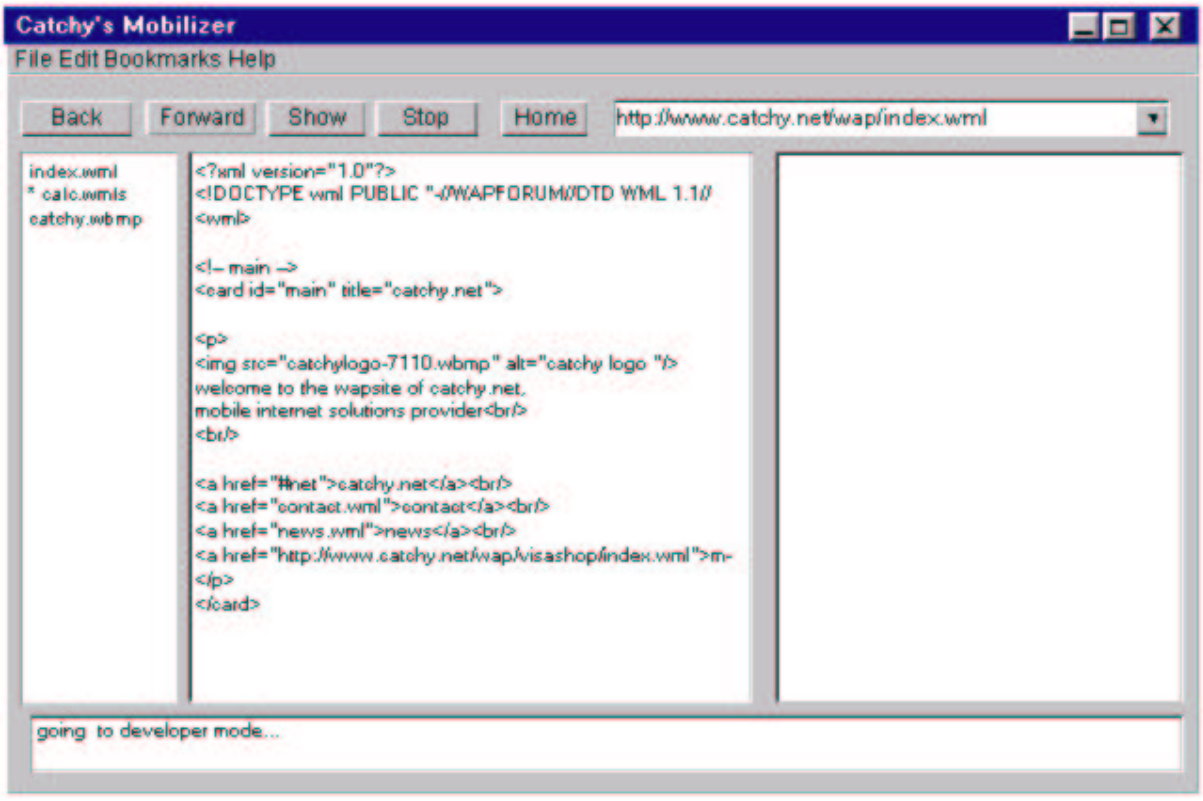

Figure 3-2, developer mode

There are now two new windows: The left-most window which shows all open files, and a window which shows the WML code of the file that is currently in edit mode. Notice that the 'Reload' button is now the

'Show' button. Pressing this buttons will cause the program to render the WML code in the right-most window.

4. Mobilizer, Chronological Story

In this chapter the whole story of the development of the WAP-browser during the apprenticeship is presented in chronological order. Every iteration is split up in four parts: functional specification, technical specification, tasks and implementation. At the beginning of every iteration the XP team decides what is going to be build in the coming iteration. The 'functional specification' part tells what this is and how this was decided. The 'technical specification' part describes the design, the 'tasks' part tells in which parts the work was split up and the 'implementation' part tells the things that happened during the implementation.

Starting Up

Day 1 - 7
In this period a couple of things had to be done. First of all, the team had to become familiar with some tools. We set up a *CVS* server to make integration possible, a Microsoft Visual Studio project was made and we started using javadoc, a tool that makes HTML documentation from c++ source code. We have set up a homepage for the WAP-browser on the intranet of Catchy, so that we were able to let colleagues know what we were up to. On this homepage we published the first version of the coding conventions. We also wrote a functional specification of the final product we had in mind. This text can be found in chapter 3 of this document.
There was some discussion about whether we should make our own WML parser or that we should use a parser generator like flex/bison. So we tried a bit of WML parsing with flex/bison and discovered that it was very difficult to make a BNF for a mark-up language like WML. So we decided to write our own parser.

Iteration 1: WML parser

Functional Specification (days 8, 9)
When we had to decide what we wanted to build in the first iteration we could think of a couple of options: a WML parser, a WML renderer or a HTTP module. Because a WML renderer could not be tested without a WML parser and a HTTP module would not be very useful without the WML renderer, we decided to start with a WML parser.
The functional specification of the first iteration was very simple. It consisted of figure 4-1 and a description of the program. In the left text box the user could type text and when the 'parse' button is pressed, the text is parsed and when no parse errors are found the parse tree is displayed in the right window, otherwise an error text is displayed.

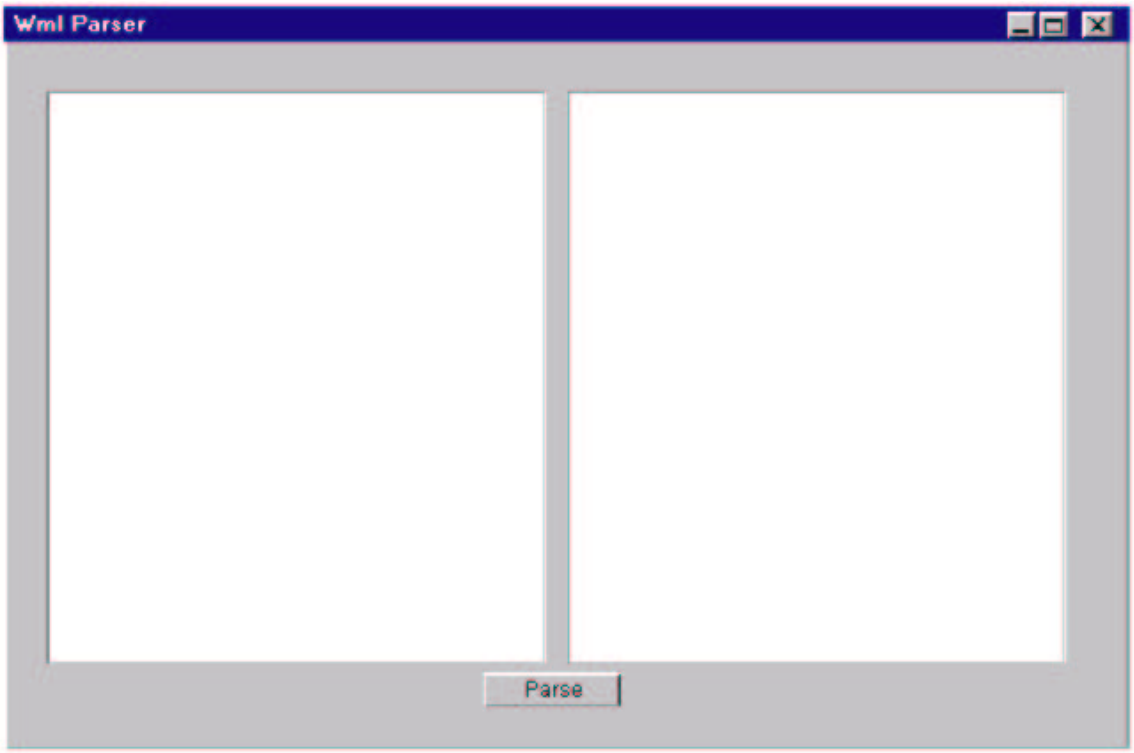

Figure 4-1, the user interface from the functional specification of the first iteration

Technical Specification (day 10)

The technical specification is more difficult. The parsing starts with an object that is an instance of a class called CharStream. This class makes it possible to walk through the WML source; it has methods like getNextChar(). The XMLTokenizer uses this object. This class tokenises the WML source. XMLTokenizer is used by XMLParser, which builds a parse-tree from the tokens. This parse-tree consists of XMLElement, XMLAttributes and XMLBody objects. Finally WMLParser checks the parse-tree with the DTD of WML, in other words it checks whether all tags are in the right place and have the correct attributes.

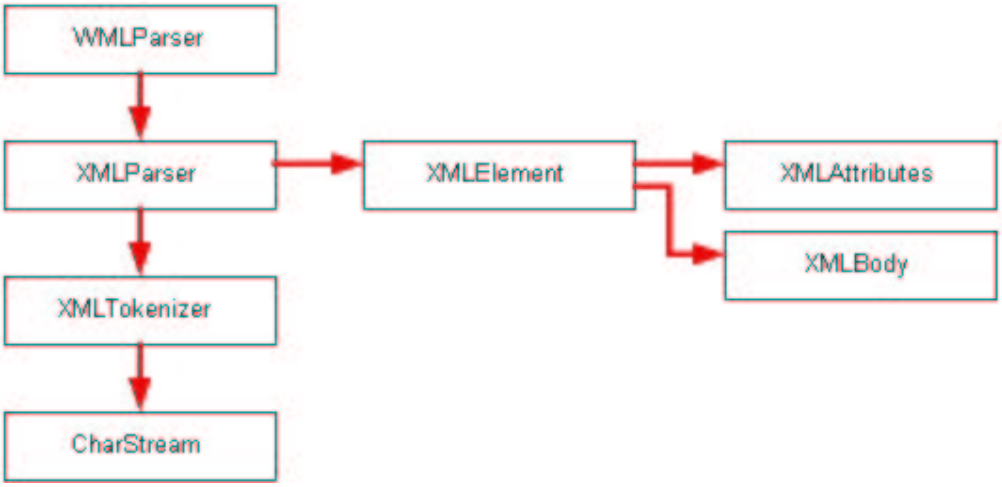

Figure 4-2, design of first iteration

The parse-tree is best explained by means of an example. The following piece of XML code would result in the parse-tree in figure 4-3.

```
<example attribute1='true' attribute2='false'>
 bla bla
 <example2/>
</example>
```

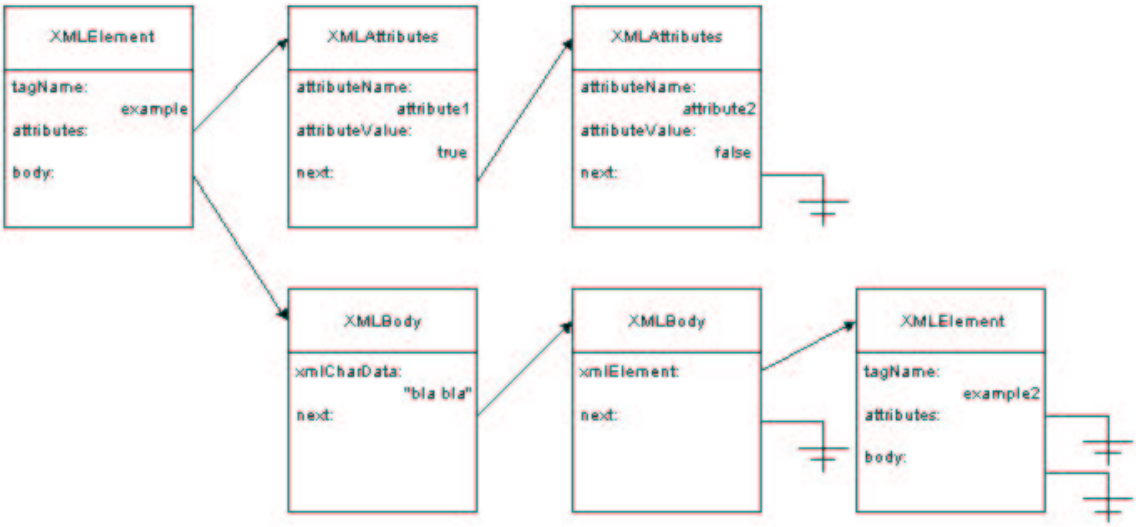

Figure 4-3, a parse tree example

Tasks (day 10)

The list of tasks looked like this:

Task	Estimated duration in days	Real duration in days
Implementing class String	1	1
Implementing class CharStream	1	1
Implementing class XMLTokenizer	1	1
Implementing class XMLAttributes	1	0.33
Implementing class XMLBody	1	0.33
Implementing class XMLElement	1	0.33
Implementing class XMLParser	1	1
Implementing class WMLParser	3	4

Implementation (day 11 - 19)

The implementation of the WML parser went according to plan. The deliverable can be seen in figure 4-4. Notice that the output is indented to make it readable for humans. Also notice that the attributes that have a default value, like the 'optional' attribute in the 'do' tag (line 6 in the source), are put in the parse tree with their default values while these attributes were not present in the source. This was not in the specification, but was programmed as 'extra'. A second example of an 'extra' was somewhat more work. The template attribute in WML is used to code events and actions that are the same for all cards. To make things easier we decided to put the template code in the parse-tree for all cards. This is shown in the last two lines of output in

figure 4-4. Here the code of the template (lines 5-9 in the source) can be found in the parse-tree as part of the first card.

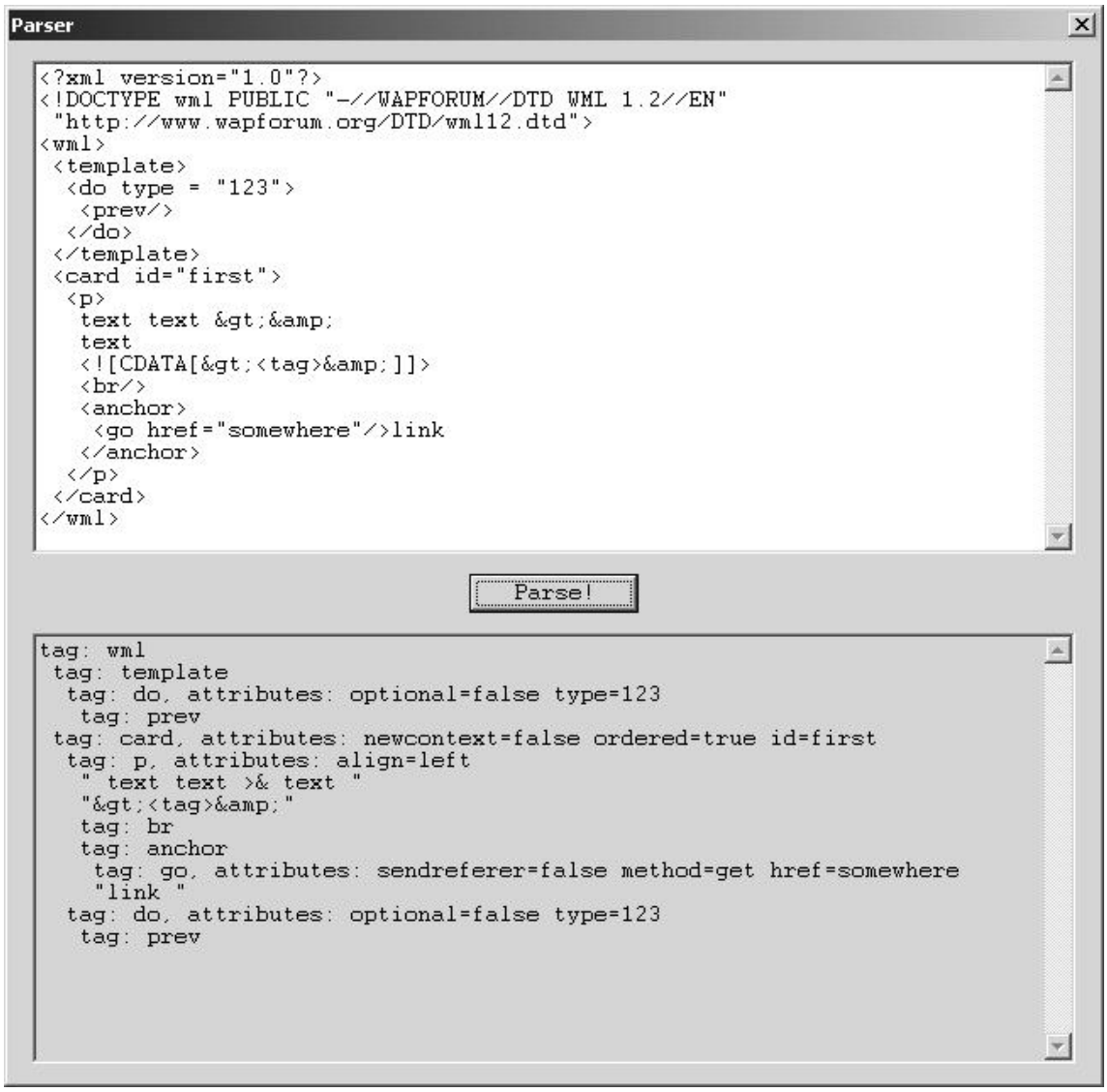

Figure 4-4, deliverable of iteration 1

Iteration 2: <do> tag

Functional Specification (day 20, 21)
We decided that the next thing to build was a renderer. At first we thought that the whole renderer should be build in one iteration, but after trying some things it became clear that it was a lot more work than we thought. So we split the renderer into a few iterations.
The first iteration consisted of a few things. I made one of them in the specification period to see whether it was possible. It was the idea of a program with a child window as in figure 4-5. The program centralizes its child window when the child window or the main window is resized. This child window will be used to display the WML content in the coming iterations.

Figure 4-5, from the functional specification of the second iteration

Another thing that was going to be built in this iteration was the displaying of the <do> tags. [WML] says about the do element:

> The do element provides a general mechanism for the user to act upon the current card, i.e., a card-level user interface element. The representation of the do element is user agent dependent and the author must only assume that the element is mapped to a unique user interface widget that the user can activate. For example, the widget mapping may be to a graphically rendered button, a soft or function key, a voice-activated command sequence, or any other interface that has a simple "activate" operation with no inter-operation persistent state. When the user activates a do element, the associated task is executed.

Gerald Stap, R&D manager of Catchy at the time, suggested to display the do elements as the drop-down select box in the lower left corner of figure 4-5. The idea is that all the do elements of the current card are displayed in the select box and that the user can choose from the list and then can activate one of them with the button under the select box. And the last thing for this iteration was the integration of the parser in this program. Parse errors (if present) were going to be displayed in the text box at the bottom of the window (see figure 4-5).

Technical Specification (day 22 - 25)

During the making of the technical specification we had to decide whether the WAP-browser was going to be multi platform, because we were going to connect platform independent code (the WML parser) to platform dependent code (the dropdown menu, parser error box). We decided to make

the WAP-browser multi platform, starting with a Windows and a Linux version. This meant that a platform dependent class had to be made which controlled the drop down box, which could have different implementations for different platforms.

Tasks (day 25)
The list of tasks looked like this:

Task	Estimated duration in days	Real duration in days
Implementing resizable window	2	2
Implementing file loading code	0.5	0.25
Integrating WML parser	0.5	0.25
Connecting WML parser to the parser error box	0.5	0.25
Connecting WML parser to the dropdown menu of <do> elements	0.5	0.25

Implementation (day 26)
This iteration consisted of very basic implementation (file loading, displaying text in windows, ...). This kind of implementing is relatively easy to plan; we were able to finish this iteration within the deadline.

Iteration 3: rendering text

Functional Specification (day 27)
In this iteration we wanted to let the WAP-browser display text. All other things present in the WML-page should be ignored. When the text is too big to fit in the window, a scrollbar should appear, this could be a vertical scrollbar (when there are too many lines to fit on the screen) or a horizontal one (when a word or a line that may not be broken into multiple lines does not fit on one line).

Technical Specification (day 27, 28)
The specification of this iteration was mainly done in the specification period of the second iteration. The main problem is to keep the program platform independent. The specification was based on the idea depicted in figure 4-6.

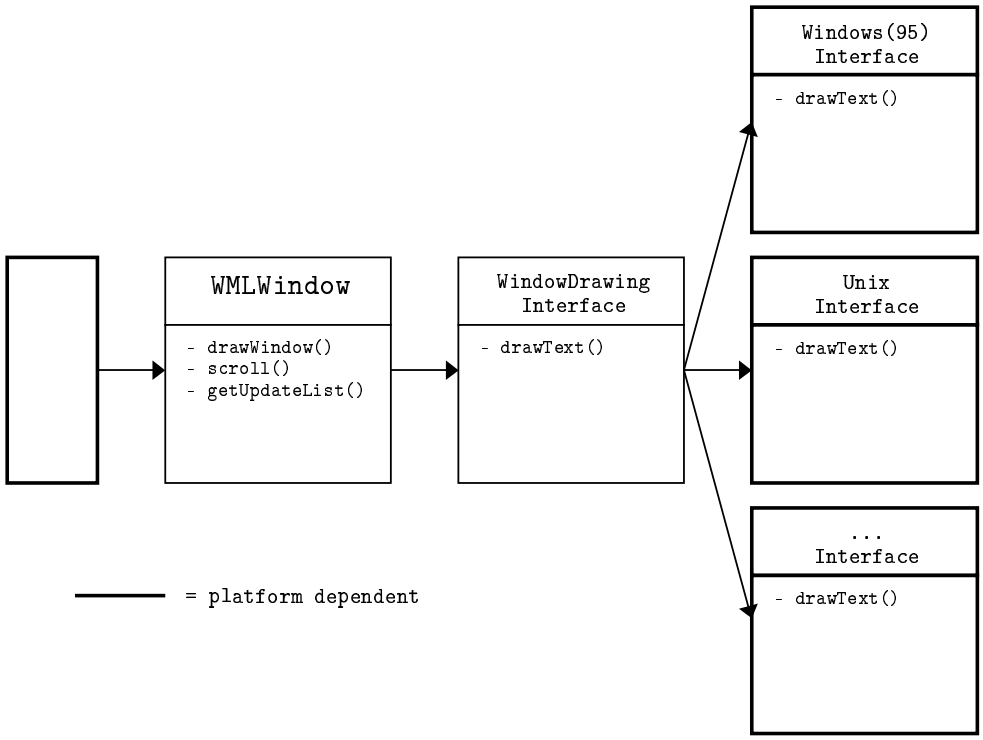

Figure 4-6, platform independent rendering

The box on the left is the code that detects that the window has to be drawn. It then calls the drawWindow member of the WMLWindow class. When text must be drawn in the window it calls the platform dependent drawText member of the correct interface. These interface classes are all derived from the abstract WindowDrawingInterface class.
When a scrollbar must be shown or positioned or when a do element has to be placed in the select box, WMLWindow places these instructions in a so-called update list. The platform dependent code can access this list by calling the getUpdateList member of the WMLWindow class.

Taks (day 28)

Task	Estimated duration in days	Real duration in days
Make WMLWindow, WindowDrawingInterface, WindowsInterface, GNOMEInterface	1	1
Normal text displaying	1	2
Enable text of different sizes	3	5
Enable different alignments	3	10

Implementation (day 29 - 46)

This iteration was finished two weeks after its deadline. The tricky thing about rendering text is that it is very easy to display left aligned normal text, but things get very difficult when you throw in right aligning and fonts with different heights. The trouble with right- or centre- aligning is that you don't know the x-position of the first word on a line when you don't know the total length of the words on that line. The trouble with fonts of different heights is that you don't know

the y-position of the first word on a line when you don't know the maximum height of the words on that line, in other words: when the second word on a line is bigger than the first, the first word has to be displayed lower.
Another problem that came on our path was the rendering of the following bit of WML code:

```
<wml>
<card>
<p>
<b>bold</b> <i>italic</i>
</p>
</card>
</wml>
```

Should this code be rendered as '**bold** *italic*' or as '**bold***italic*'? The problem here is that the XML specification ([XML]) is not clear on whether the space, which is located in a place in the code where character data is allowed, should be treated as character data or as whitespace. The same holds of course for the other whitespace characters in WML (carriage returns, line feeds and tabs). A short investigation among the popular WAP-browsers at the time showed the following:

With space	Without space
WinWAP 3.0	YoSpace
M3Gate	Nokia 7210
	Nokia Emulator
	UP.browser
	Sony CMD-Z5

So we decided to go with the flow and not display the space. In web-browsers the space is usually displayed (IE5, Netscape).

The Code Review

Day 47 - 59
Unfortunately we were unable to test 'pair programming' during the apprenticeship. One of the benefits of pair programming is the fact that a code review of all code takes place. We decided to also do code reviews of all code, but not during the programming. After the 3rd iteration we took some time to study each other's code and suggested some possible improvements.
The improvements were mainly in the following areas:

- Documentation - the lack of good comments which explain the code
- Unclear code - functions that are too long and therefore not easy to understand
- Memory leaks - memory that is allocated, and never freed

Iteration 4: rendering non-text

Functional Specification (day 60)
Naturally, the next iteration was going to render non-text:

- Anchors - the <a> elements must be rendered, these are the equivalents of the <a> tags in HTML. The text between the <a> and

 tag must be underlined and coloured blue. The mouse pointer must change to a hand when it hovers over the anchor.

- Input elements - the input elements must be rendered, and the user must be able to put text into it.
- Select elements - the select elements must be rendered, and the user must be able to select one of the options
- Tables - tables must be rendered, which means that all the content of the table is displayed in cells, all cells in the same row must have the same height, all cells in the same column must have the same width
- Pictures - the WBMP pictures as specified in [WML] must be rendered

Technical Specification (day 60, 61)
I will only cover anchors here because the others are straightforward or are not in the scope of this document.
When the program is (re)drawing the screen it puts all coordinates of the anchors in a list, the so-called AnchorList. With every mouse move the program checks whether the mouse is located on one of the anchors, if this is the case it changes the mouse pointer into a hand.

Figure 4-7 hand mouse pointer hovers over a link.

Tasks(day 61)

Task	Estimated duration in days	Real duration in days
Anchors • Render differently • Put in AnchorList • Change mouse pointer to hand when necessary	2	2
Input elements	2	3
Select elements	2	1
Tables	2	4
Pictures	2	4

Implementation(days 62 - 75)
The implementation was all pretty straightforward, except for the implementation of tables. During the making of the technical specification we overlooked the fact that tables could also contain pictures and input and select elements, which was really difficult to implement, so after running a few days behind schedule we decided not to implement the rendering of input and select in tables.

Refactoring(days 76 - 85)
This iteration was further delayed by some refactoring that we thought was necessary. We noticed that a few classes were present in the code that were all lists and were all very similar, so we spent some days refactoring the code which made the code smaller and a (little) bit faster.

5. Mobilizer, Results

This chapter describes the results we achieved after 85 days of developing.

After 85 days of developing the windows version of Mobilizer looked as figure 5-1 shows.

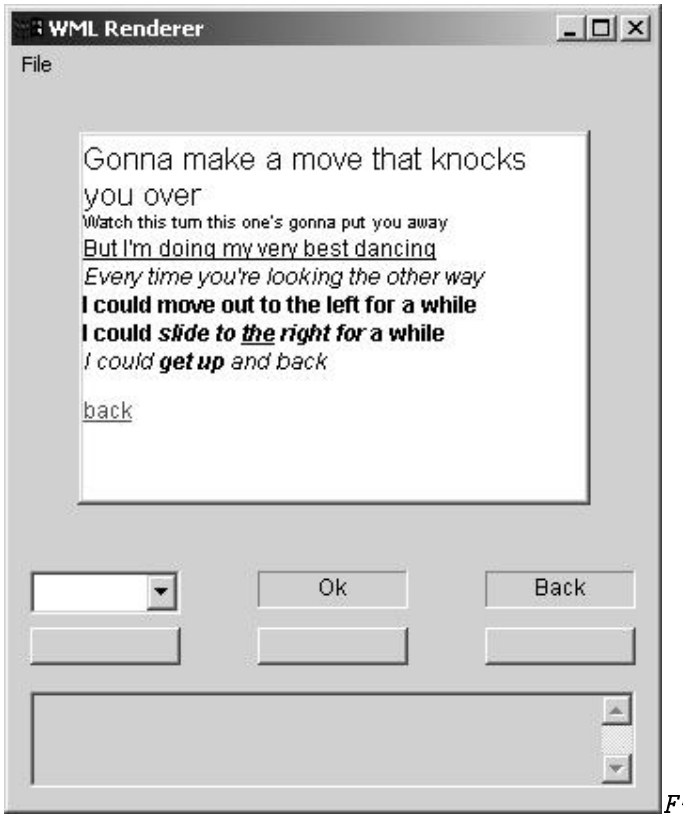

Figure 5-1

The Mobilizer in figure 5-1 is rendering the following code:

```
<?xml version='1.0'?>
<!DOCTYPE wml PUBLIC '-//WAPFORUM//DTD WML 1.1//EN'
'http://www.wapforum.org/DTD/wml_1.1.xml'>
<wml>
<card id='style'>
<p align='left' mode='wrap'>
<big>Gonna make a move that knocks you over</big><br/>
<small>Watch this turn this one's gonna put you away</small><br/>
<u>But I'm doing my very best dancing</u><br/>
<i>Every time you're looking the other way</i><br/>
<b>I could move out to the left for a while</b><br/>
<b>I could <i>slide to <u>the</u> right for</i> a while</b><br/>
<i>I could <b>get up</b> and back</i><br/>
<br/><a href='main.wml'>back</a>
</p>
</card>
</wml>
```

The growth of the system is depicted in figure 5-2.

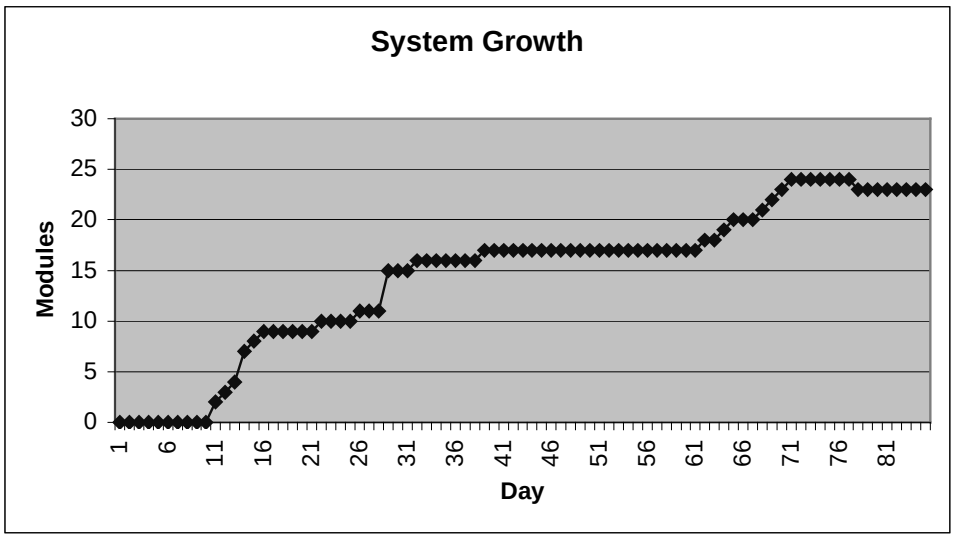

Figure 5-2

The plot shows how the amount of modules changed during the project; in this case one module is one platform independent class. The four iterations are all more-or-less recognizable. Further conclusions from this figure are left to the imagination of the reader.
To determine the productivity of the team I used a method called 'backfiring'. Productivity is usually measured in Function Points per person-month. The counting of Function Points is a difficult task and is best performed by a professional. Backfiring, as described in [BCLFP], is much easier. [BCLFP] provides, for a number of languages, the amount of source code statements per function point. For C++ this number is about 53; this is the mean value, it could be anywhere between 30 and 125.
After 85 days the Mobilizer project consisted of 2186 source code statements (determined by counting the number of semi-colons). Backfiring tells us that this is about 41 function points (between 17 and 71). 119 person-days had been invested in the project; this is 6.0 person-months. The productivity, thus, was about 6.8 function points per person-month (between 2.8 and 11.8). The U.S average is 5 function points per person-month (source: [ACSR]).

Evaluation

The initial plan consisted of four people working on this project. Sadly, only two people ended up being assigned to it, and only one full-time. This, of course, had some consequences on my research. 'Pair Programming' could not be tested, and there is little I can say about the XP practices 'Collective Ownership' and 'Continuous Integration'. To further discuss XP I decided to combine the data of the development with some articles I found in the literature. They get discussed in the following three chapters, which each focus on a software engineering theme.

6. Team Structure

To further talk about the results of this research I have chosen three software engineering themes that are discussed in the following three chapters. First I talk about the views given in a well-known book about the subject. Then an overview is given of the practices within XP concerning the software engineering theme and they are compared with the literature. After that I discuss the experiences we had during the project with the particular subject. In this chapter we focus on team structure.

Literature

A well-known example in the literature about team-organisation was proposed by Harlan Mills in 1971 and his views were included in probably the most famous software engineering book: 'The Mythical Man-Month' ([MMM]), which was published in 1975.
The problem with managing programming-teams is the dilemma between a small team and a large team. Often a small team lacks productivity, and a large team will always have communication problems. These communication problems are due to the fact that more programmers means more communication, and more communication means less actual programming, and therefore 10 programmers will never be 5 times faster than 2 programmers doing the same job. But things get worse, because more communication means also more miscommunication, which is disastrous for software engineering. An example of miscommunication that often takes place is when the code of one programmer has to be 'glued' to the code of another programmer. The interface that serves as an agreement between the two programmers has to be totally clear for both, because a slight misunderstanding can be the cause of hours of debugging, as these errors are very hard to find.
Summarizing: a small team is in most cases not capable of making the dead-line, and a large team can't work effectively, and by that is more expensive than a small team.
Mills's proposal tries to solve this dilemma between small and large teams by taking a team of 10 people and deliberately reducing the need to communicate by letting the team members specialize in a part of the job. The communication patterns in the team are depicted in figure 6-1.

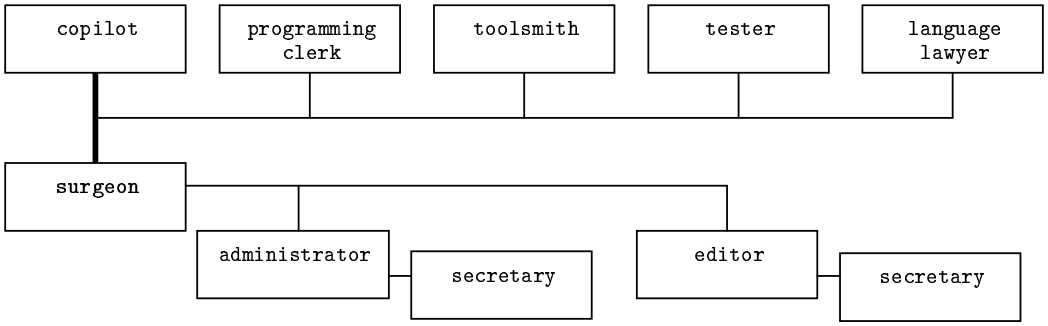

Figure 6-1, communication patterns in a 'surgeon' team

Table 6-2 shows what the tasks of each team-member are.

Team-member	Tasks
Surgeon	(Or chief programmer) Defines functional and performance specifications, designs the program, codes it, tests it, writes the documentation.
Co-pilot	Gives advice concerning the design, researches alternatives for the design, represents the team in meetings with other teams, and is the backup for the surgeon
Administrator	Handles money, people, space, machines and is the contact to the administrative machinery of the organisation
Administrator's Secretary	Assistant to the administrator, also: project correspondence, non-product files
Editor	Takes the documentation of the surgeon and criticizes it, reworks it, provides it with references and bibliography, does version control and oversees the mechanics of production
Editor's secretary	Assistant to the editor
Programming clerk	Maintaining all the technical records of the team in a programming-product library
Tool smith	Constructing, maintaining and upgrading of special tools needed by the team
Tester	Devises tests from the functional specification and test data for debugging
Language lawyer	Does small studies on good coding technique

Table 6-2

So all the 'real' work is done by the surgeon and his co-pilot, the other team-members help them by taking as much work out of their hands as possible. The surgeon and the co-pilot work together in some extend: the surgeon does all the implementation and the design is done together, with the surgeon having the last word. With this approach two problems are solved: there is no chance of miscommunication between two programmers, and no time is lost on compromising on who implements what. The problem of how to organise a team that consists of let's say 100 programmers is not yet solved, however. A large part of 'The Mythical Man-Month' addresses this problem. I will not describe it here, because XP is designed for small or medium sized teams, and therefore a comparison with XP could not be made.

XP

XP uses pair programming, which means that all code is written by two people sitting behind one terminal. They, however, have different roles. One has the keyboard and is thinking about the code he/she is writing and the other one is thinking about the 'big picture', for example: 'is this code going to work?' or 'can we simplify this code?'
A pair of programmers commits to a so-called task. This task comes from a pile of tasks that the programmers can choose from. When the task is finished the programmers can choose to work with someone else on a new task.

A programming team has, besides normal programmers, also two special members: a tester and a tracker. Often they will also be normal programmers beside their special functions.
The main function of the tester is to run all tests regularly and communicate the results to the rest of the team. The tester may also have the responsibility of helping the customers write tests, as they generally won't have programming skills.
The tracker gives feedback to the team about their estimates.

XP vs. Literature

Mills's 'surgeon team' almost guarantees conceptual integrity, because one programmer writes all code. But this programmer must have some exceptional capabilities, as he must provide work for 9 people while developing the program.
The XP-approach must be the exact opposite of the 'surgeon team'. Instead of every team-member specializing in something, no one specializes and every team-member is equal, which minimizes the risk of a project depending on a few people. A disadvantage of the XP-approach is the fact that programmers tend to dislike pair programming, which became apparent when XP was first suggested to the programming team.
When one must choose between the 'surgeon team' and XP, I would only suggest the 'surgeon team' in cases where one has the disposal of one exceptional programmer and conceptual integrity has very high priority. In other cases, XP must be a better choice.

Mobilizer

Sadly I could not test pair programming in this project because of conflicting schedules within the team. But we did manage to have a code review of all code.
We found working with tasks to be comforting. It gives a feeling of how much progress the iteration is making, and also gives a feeling of direction while programming.
The tracker of our team discovered that our estimates were too low in the first two iterations and too high in the last two. A possible explanation can be found in the next chapter.

7. Lifecycle

In chapter 6 we focussed our attention on team organisation ('how to divide the work among people') because we know we cannot let a software engineering project be done by one person (in theory the most effective team). We also know that the project won't be finished in one day, so in this chapter we look at how the work should be divided among the time-line: the lifecycle.

Literature

In his book 'Rapid Development' ([RD]) Steve McConnell describes several lifecycle models. I will discuss seven of them here, those that I found most important and relevant.

Pure Waterfall

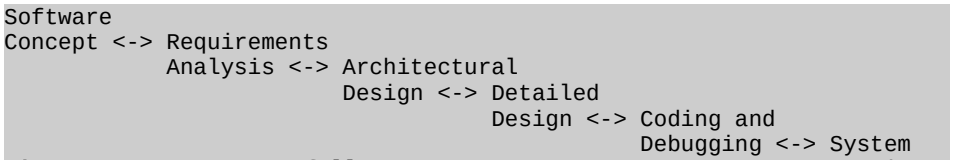

Figure 7-1. *Pure Waterfall.*

Figure 7-1 shows how the 'Pure Waterfall' model works.
There are 6 stages in this model, the project starts with the software concept stage and ends with the system testing stage. To advance from one stage to the next the team holds a review, so it can determine whether the project is ready or not.
The main disadvantage of this model is the troubles that arise when the requirements appear to be incomplete whilst the project is for instance amidst the coding and debugging stage. It is allowed to go back a stage, but it being a waterfall makes this very hard: changing requirements changes the design and can make already programmed modules become obsolete or they must be redone. So these troubles waste time and money, and this is a big problem because the incompleteness of requirements is very common in software engineering.

Sashimi

In the pure waterfall model all stages are disjoint, thus making it possible to hand the project to a completely separate team between any two stages. If personnel continuity is sufficiently present one can allow the stages of the pure waterfall to be overlapping, this makes the transitions from stage to stage more efficient. This is the sashimi model ('sashimi' refers to the Japanese style of slicing fish, with the slices overlapping each other). A disadvantage of this model is the fact that parallel activities in the different stages can lead to more miscommunication.

Waterfall with Subprojects

When the system that must be built consists of a number of mostly independent subsystems this model can be efficient. Figure 7-2 shows the concept of this model.
After the architectural design stage is done several subprojects are started and each of those can proceed on its own pace. After all subprojects are done all of the code has to be integrated into one program, and that program must be tested as a whole.

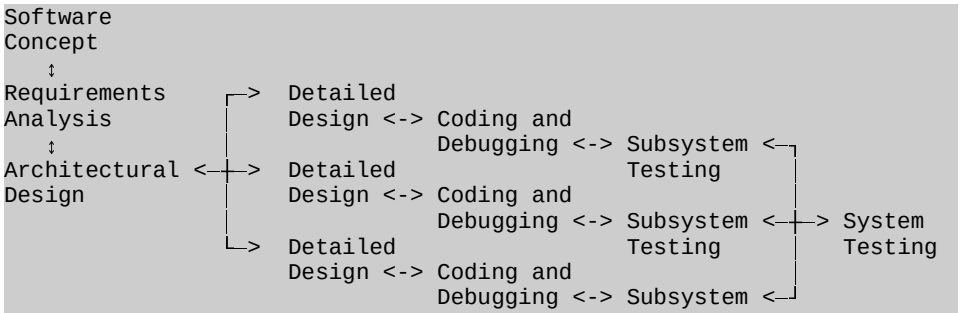

Figure 7-2. *Waterfall with subprojects.*

The advantage of this model is that some of the programmers can already start with one of the subprojects while others are still busy with the architectural design, which generally is a stage in which only a few people are involved. So this model can be more effective than the pure waterfall model.
The main problem with this approach is the risk of unforeseen interdependencies; subprojects could start prematurely.

Evolutionary Prototyping
The last three lifecycle models are all iterative, which means that there is a loop in the lifecycle. The loop of the evolutionary prototyping model is displayed in figure 7-3.

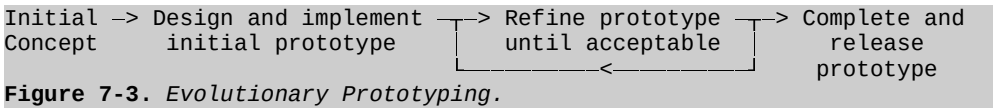

Figure 7-3. *Evolutionary Prototyping.*

From the initial concept of the program a prototype is build with most attention paid to the visual aspects. Then based on the feedback from the customer the prototype is developed further. This process continues until the customer is fully satisfied.
This model is useful when the requirements are rapidly changing or there is little understanding of the application area. Disadvantages are the fact that it is hard to give an estimate on how long the project will take, and this method has increased risk for bad ('spaghetti') code.

Staged Delivery
The staged delivery model works a lot like evolutionary prototyping, there are however two major differences. As figure 7-4 suggests, with the staged delivery model one goes through some steps of the waterfall in the iteration. After each iteration the program is delivered to the customer who, contrary to the evolutionary prototyping model, can start to use it. His feedback is also welcome.
The main advantage is that one can give valuable functionality to the customer in an early stage of the lifecycle. Disadvantages include the difficulty of carefully technical planning; imagine the possibility of planning a component for iteration 4 only to find that a component for iteration 2 can't work without it.

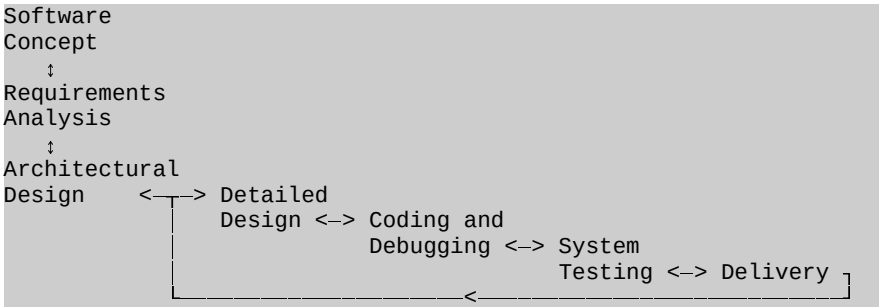

Figure 7-4. *Staged Delivery.*

Design-to-Schedule

The design-to-schedule model is almost the same as the staged delivery model, the only difference is that all the iterations have priorities, and the iterations are done in order of priority: highest priority first, lowest priority last. Advantage: big chance there is a good product when the dead line comes. Disadvantage: possibility of wasting time on architecting features that will never be implemented.

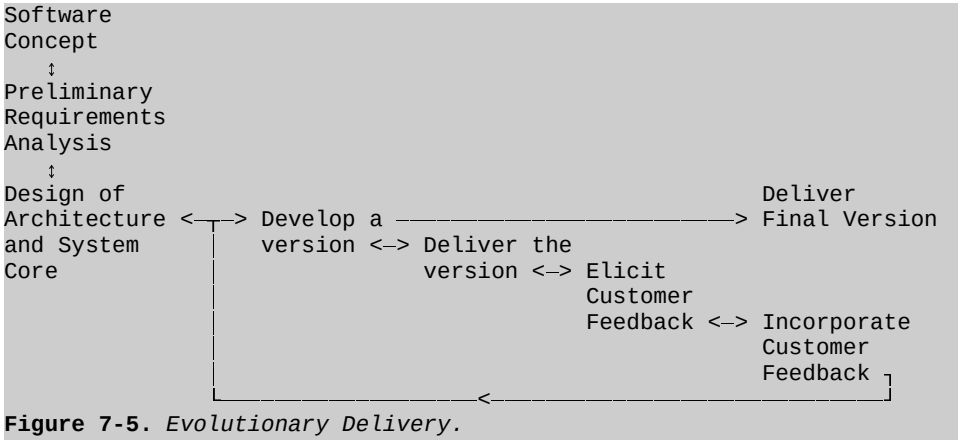

Figure 7-5. *Evolutionary Delivery.*

Evolutionary Delivery

The evolutionary delivery model (figure 7-5) looks a lot like the evolutionary prototyping and staged delivery models. The difference with evolutionary prototyping is that with evolutionary prototyping the initial emphasis is on the visual aspects of the system, with evolutionary delivery this isn't necessarily so. Staged delivery pays less attention to customer feedback than evolutionary delivery.

XP

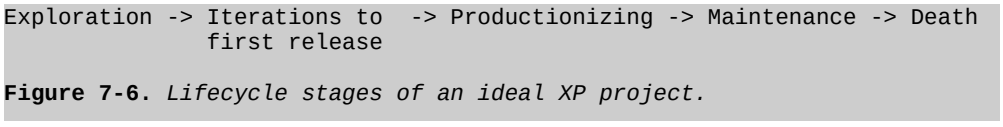

Figure 7-6. *Lifecycle stages of an ideal XP project.*

The stages of the lifecycle of an ideal XP project, described in chapter 21 of [XP], are depicted in figure 7-6.

The first stage is the exploration stage, in this stage the programmers explore the possibilities for the system architecture, so that the team has an idea how the architecture should be when the developing starts. The planning and committing to the planning is very important in XP, so

in the exploration stage the programmers also practice in giving good estimations on their programming speed.
The next stage is called 'iterations to first release'. In this stage all the functionalities of the first release are build in iterations. The first iteration will put the architecture in place. Next iterations will each build the most important (business-wise) functionalities left to build.
The iterations have two sub-stages: the 'planning' stage and the 'developing' stage. In the planning stage the business people and development people decide what is going to be build by means of writing stories. They also estimate how much time those stories will take to develop and how much time the whole iteration will take.
In the developing stage the stories are translated into tasks. The programmers subsequently write, accept, estimate and implement the tasks. Implementing a task consists of writing test cases for it, writing the code and verifying that the test cases work. When all the tasks are done the team verifies that all stories are implemented.
When all the stories of the first release are implemented the project can go to the 'productionizing' stage, which means that the product is about to be shipped to the customer(s). In this stage the product is tested more thoroughly and also some performance tuning is done.
After the first release the project goes into the 'maintenance' stage. Besides developing new functionalities the team must now also react on feedback of customers, this can be removing bugs or working on the help desk. Productivity will therefore be lower than before going into production. In this stage the team also works in iterations.

XP vs. Literature

The lifecycle of an XP project is a lot like the last four lifecycle models discussed in the 'Literature' section. The main difference is that with XP for every iteration an architectural and a detailed design are made for that iteration. Hereby an XP project deliberately takes the risk of having to heavily alter the architecture of already written code because a new feature can't work with that architecture. The question, of course, is how big this risk is. It could well be, and XP is betting on this, that the risk is not much bigger with XP than it is with the conventional lifecycle models, as it is practically impossible to make a flawless architectural design before a line of code has been written.
The advantage of XP is that when the architecture must be changed amidst a project, an XP team can more easily handle this situation. Team-members are not fixed to one thing, but they are used to do different things with different people because they work with tasks and pair programming. And they are used to changing code, as they are used to refactoring.
So I conclude that the more difficult it is to make the architectural design without mistakes the better it is to use the XP lifecycle model.

Mobilizer

In the planning stage of the iterations we wrote a document with all the stories that we called the 'functional specification'. Gerald Stap, R&D manager of Catchy, played the part of customer. By making the specification this way we were able to be very specific on what was

going to be build, much more specific than when we had to make the specification for the whole program.
When one wants to make a decision on whether the XP lifecycle model is a good one, one first has to decide on how one decides that a lifecycle model is a good one. I think that the main function of a lifecycle model is to keep the project manageable, in other words, to provide the 'outside world' with sufficient information on how the project is developing. The most important information for managers is the date on which the project will be finished. Let's take a look at the schedule of the Mobilizer and by how much they were missed:

Iteration	Estimated duration in days	Real duration in days	Exceeding planned date in %	Reason
Iteration 1	10	9	-10%	
Iteration 2	4	3	-25%	
Iteration 3	8	18	+125%	Underrated problem
Iteration 4	10	24	+140%	Refactoring

From this table one can see that the estimates were too pessimistic in the first two iterations and (far) too optimistic in the last two. This could be due to the fact that the first two iterations were smaller, but is probably also because the last two were more complicated, as the code of the latter iterations must communicate with the code of the earlier iterations.
[ACSR] states that the average exceeding of the planned date is between 25% and 50%. It further notices that the risk of missing schedules increases when the project:

- Is relatively big
- Is relatively new in concept
- Has inexperienced management and staff
- Doesn't use planning and estimating tools

Which was all more or less the case with the Mobilizer project.
Interesting is also that [ACSR] states that projects are more likely to miss their schedules at a later stadium of their development, which also happened with the Mobilizer.
One may come to the conclusion that the XP lifecycle model was not the best choice for this project, as two of the four iterations missed their deadlines by far more than the average given by [ACSR]. But when I speculate a little on how the schedule slips would be when we wouldn't have used XP, a different picture appears.
Let's say we would have made the architectural design for all four iterations at once and successively would have implemented the four iterations. The schedule slip would then have been more than the average of the four percentages above, because

1. The architecture would have been much more difficult, so it would have taken more days than the sum of the days spent on the four separate specifications.
2. The estimation of the fourth iteration was based on the experience of missing the dead line of the third iteration; the estimation of the four-iterations-at-once project wouldn't have had this experience.

8. System Integration

This chapter focuses on system integration, which is the combining of separate software components into a single system.

Literature

In 'Code Complete' ([CC]), McConnell describes a number of integration strategies.

Phased vs. Incremental

The integration can be done phased or incremental. A project that uses the phased integration follows these steps:

1. Design, code, test and debug each routine
2. Combine the routines into one system
3. Test and debug the whole system

Step 2 will inevitably introduce problems; this can for instance be caused by an interaction between two routines. The problem with this integration strategy is that these problems will come in large numbers and that the location of the problems could be in any of the routines. This makes testing and debugging an enormous effort, and that's why phased integration is only suitable for very small projects.
A project that uses incremental integration follows these steps:

1. Develop a small part of the system
2. Design, code, test and debug a routine
3. Integrate the new routine, test the whole system, and go to step 2.

This approach has many advantages over the phased integration. When new problems surface during the integration, the new routine is obviously at fault, so these problems are easy to locate. With this approach each routine is tested more, as routines that are integrated at the beginning of the project get tested with each consecutive integration.
There are several kinds of incremental integration strategies. They differ in the order in which the routines are integrated.

Top-Down Integration

In top-down integration the routines in the top of the hierarchy are integrated first. (These are the routines that are not called by other routines.) Stubs are written for the function-calls that are made from these routines. Then real routine replace the empty ones, and stubs are written for function-calls in these routines, etcetera. Finally the routines that don't call any other routines are written and integrated.
The main advantage of this approach is that design problems are discovered relatively early; a disadvantage is that low-level problems could 'bubble up' and lead to high-level changes and by that reduce the benefit of earlier integration work.

Bottom-Up Integration

Bottom-up integration works exactly the other way around: First the low level routines are implemented and than the routines that call those routines, etcetera.
The advantages and disadvantages are also the other way around: a disadvantage is that design errors are discovered relatively late, and an advantage is that low-level problems are discovered relatively early.

Sandwich Integration
To avoid the disadvantages of the top-down and bottom-up strategies one could choose the sandwich integration: First the top level is implemented, then the lowest level routines are integrated and finally the middle level(s) are integrated.

Risk-Oriented Integration
In risk oriented integration one identifies the level of risk associated with each routine. The more challenging the routine is to implement and integrate the higher the risk. The routines with the highest risks are integrated first, this way one tries to avoid that big mistakes are discovered late in the project.

Feature-Oriented Integration
With feature-oriented integration the routines are first integrated to become features, which are subsequently integrated to become the program. An advantage of feature-oriented integration is that with each feature that is integrated there is evidence that the project is moving steadily forward, which is good for morale.

XP
XP works with 'continuous integration', which means that after every task the work is integrated. The tasks are done in almost arbitrary order: there is a pile of tasks that must be done in the current iteration, and the programmers can choose a task from that pile. The iterations are done in order of business importance: the functionalities that are most important are developed first.
To ensure that the program still functions completely after integration, the programmers write test cases for all the routines. All the tests of the program are executed after each integration.

XP vs. Literature
Continuous integration seems to be a combination of the risk-oriented and feature-oriented integration strategies mentioned in [CC]. A difference is that all the strategies mentioned in [CC] have the programmers integrate a routine at a time, while XP integrates tasks.

Mobilizer
It is the opinion of the team-members that integrating the code continuously is a very good practice. One can think of non-integrated code as a risk. You don't know if it compiles, works or whether it ever will be integrated. A good example of this is when we decided to port the code to Linux. The WML-parser already consisted of quite a lot of lines code and when we compiled it with a Linux compiler we discovered that some of the files didn't compile. We had to make quite an effort to make the code compileable under both platforms. This shows that one cannot assume that code that is not frequently compiled and tested (on all platforms) will work as expected.
When some bit of code works, it doesn't mean that it will keep working. This is why XP also uses continuous testing. We decided to give each class a static method called 'test' that tests all the code of that class. Figure 8-1 and 8-2 show how this works. To make sure that every line of code is tested we used a tool that showed the *coverage* of the

```
/**
 * This function tests the code of this class
 * @return true when all tests succeed, otherwise false
 */
bool String::test() {

  String s = new String("hallo");

  if(s.getLength() != 5)
    return false;

  if ...
```

Figuur 8-1. *Sample code of class String.*

```
testing class 'String'... passed.
testing class 'CharStream'... passed.
testing class 'XMLTokenizer'... passed.
```

Figuur 8-2. *Sample output of tests.*

code, in other words, how many times each line of code was executed during the execution of the program.

9. Conclusion

In this paper I've described XP, described the development of the WAP-browser, and discussed three software engineering themes and their relation to XP and the Mobilizer project. In this chapter I will draw the conclusions of this apprenticeship.

When I compare [XP] with other software engineering books like [RD] and [CC] I notice that [XP] is in comparison very small and written in a much more light-weighted tone. The other books are much more comprehensive and use more arguments with their statements. One could assume that the intended audience of [XP] are managers and not programmers. Of course it is the decision of managers and not of programmers to use XP, but if XP wants to be taken seriously in the software development world, shouldn't it be described less sounding like a commercial?
But let us take a look at the things we learned about XP.
The team structure is very important in XP. Everybody is equal and nobody specializes in anything. The 'surgeon' team, as described in chapter 6, is exactly the opposite, here we have one programmer who does all the programming, and all other members are specialized in something to make the work of the 'surgeon' as easy as possible. This method is depending very heavily on him/her, when there is no 'super' programmer available the XP method is a good alternative.
When avoiding schedule slips has high priority, working with iterations is a good choice. As chapter 7 describes, the schedule slip of the first four iterations of the Mobilizer project would have been bigger when no iterations had been used. To further avoid schedule slips, the team should reckon with the fact that refactoring will sometimes have to take place, when determining the deadline of the coming iteration.
We have also seen that with XP the team does not have to make an architectural design of the whole program at the beginning of the project. When the project is too big, difficult or uncertain to make a complete architectural design for, the XP method is a good option.
Continuous integration in combination with continuous testing is a good method in keeping the risk of unmanageable code low (chapter 8). To make sure that every bit of code is tested one should determine the *coverage* of the code. This makes continuous testing even more powerful.
When the XP practices are compared with the practices in the literature (especially [RD] and [CC]) one can see that the XP practices are hardly new, but the combination of them is new. During the project we did not find any flaws in the XP practices.
Concluding, XP has some practices that are valuable in certain circumstances. We did not find any flaws in the XP practices.

10. Glossary

- CVS - concurrent versions system, a 'source control' tool designed to keep track of source changes made by groups of developers working on the same files, allowing them to stay in sync with each other
- Coding conventions - a collection of agreements on how to write code, like how to write comments, how to indent, etc.
- Code review - the process of letting someone other than the person who wrote the code have a look at the code and suggest or make improvements
- Coverage - the coverage of a piece of code - how many times each line has been executed during the execution of the program
- DTD - Document Type Definition, of every XML language the DTD tells what *tags* are allowed where and which attributes they must/may have
- Iteration - a one- to four- week period in which a specific functionality is added to the program
- Release - the program that is released to an audience
- Requirements - a text that is written about a program before it is developed, it states what the program must look like and what it must do
- Story - a text in which a single feature of the program is specified
- Tag - elements in XML, tags start with '<' and end with '>', for example: <tag attribute='value'>
- Task - an assignment for a programmer, this could be implementing a story, rearrange a bit of code, etc.

11. References

[ACSR] "Assessments & Control of Software Risks", C. Jones, Yourdon Press, 1994

[BCLFP] "Backfiring: Converting Lines of Code to Function Points", C. Jones, *IEEE Computer 28*, 11 (November 1995).

[CC] "Code Complete: A Practical Handbook of Software Construction", S. McConnell, Microsoft Press, 1993.

[HTML] "HTML 4.01 Specification, W3C Recommendation 24 December 1999", D. Raggett et al, December 24, 1999. URL: http://www.w3.org/TR/REC-html40/

[RD] "Rapid Development: Taming Wild Software Schedules", S. McConnell, Microsoft Press, 1996.

[MMM] "The Mythical Man-Month: Essays on Software Engineering", F. P. Brooks, Addison-Wessley, 1975, 1995.

[WML] "Wireless Markup Language Specification", WAP Forum, 04-November-1999.

[XML] "Extendible Markup Language (XML), W3C Recommendation 10-Febrary-1998, REC-xml-19980210", T. Bray et al, February 10, 1998. URL: http://www.w3.org/TR/REC-xml

[XP] "eXtreme Programming, Embrace Change", K. Beck, Addison-Wessley, 1999