Developing Software With Extreme Programming Using Some Tools of Kanban

A Field Study

Christoph Friedl

Master's Thesis

Graz University of Technology

Adviser: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Statutory Declaration

Abstract

Der Einsatz agiler Softwareentwicklungsmethoden ermöglicht es, Software mit dem Kunden im Fokus zu produzieren. Auf die Wünsche des Kunden kann nur dann ausreichend eingegangen werden, wenn man die Möglichkeit hat, das entstehende Produkt jederzeit auf sich ändernde Anforderungen anpassen zu können.

Die kompromissloseste Form von agiler Softwareentwicklung ist "Extreme Programming" (XP). Ein XP-Team kann sich sehr agil bewegen und sich schnell neu ausrichten. Außerdem werden das Qualitätsbewusstsein und schnelle Entwicklungszeiten gefördert.

Eine weitere agile Methode wird Kanban genannt. Hier stehen Transparenz und das schnelle Erkennen von Engpässen im Vordergrund. Die aus Japan stammende Methode wurde zuerst in Industriebetrieben verwendet, ist aber auch in die Softwareentwicklung übernommen worden.

Aus diesen zwei Methoden wurde eine neuartige Entwicklungsmethode gebildet. Genauer gesagt, wurde XP mit mehreren Elementen von Kanban versehen. Die Praxistauglichkeit dieser Methode wurde im Zuge einer Lehrveranstaltung der TU Graz im Sommersemester 2011 getestet.

Abstract

The application of agile software development methods allows developing software in a customer-centric way. Changing requirements in a software project can only be fulfilled if there is a way of adopting the project to these changes.

The most uncompromising way of agile software development is Extreme Programming (XP). An XP-team can move in a very agile way and head into a new direction. High software quality and fast development are supported.

Another agile development method is called Kanban. It focuses on transparency and the fast detection and avoidance of bottlenecks. This method introduced in Japan was at first used in industrial manufacturing environments. Nowadays, Kanban has also been adopted to be used in software development.

These two methods were combined which resulted in a new software development process. The practicability of this process was tested in a class at Graz University of Technology in Summer 2011.

Table of Contents

1. Introduction

Often people ask me what I do in my job. I answer them that I work as a software engineer. After a few seconds of thinking they often come up with: "Ah, so you are a programmer". I cannot tell that this is a wrong answer but it is a true answer neither. Software engineering is a lot more than just programming.

The first thing to start with in this thesis is to give a definition of software engineering. According to Abram, it is defined as follows:

"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software." (Abram 2004)

The history of software development is as old as computers are. Over the time, standardized methods were developed to describe how software should be created. Often a long time passed from the beginning of a software project to the delivery at the end of the project. Also often the customers did not like then what they got then. The result was unsatisfied customers, a lot of change requests, which often nearly took as long as the first implementation period. Even the cancellation of a project was not unusual. The two main reasons for such an outcome are inflexible software development processes and the lack of important customer feedback during the development phase. This is the point were agile software development methods come in.

The use of agile development methods is getting more and more interest in software business. Flexible methods like Scrum substitute traditional processes like waterfall or V-model. The scope on the customer when producing software is an important fact when it comes to survive in a competitive environment. The wishes of customers can only be fulfilled if the developers can adopt the software to be built to changing requirements.

As the word "agile" already tells, such development methods help to change the direction a software project is heading to. Agile software development makes it possible to change project specifications with keeping the effort at a minimum.

The most uncompromising form of agile software development is called "Extreme Programming" (XP). A lot of rules make this process quite strictly regulated. Nevertheless, an XP team can move in a very agile way through following these rules. High software quality and fast development are the result.

Another agile development method is called Kanban. Its focus is set on transparency and the fast detection and avoidance of bottlenecks. This method introduced in Japan was at first used in industrial manufacturing environments. Kanban has been adopted to be used in software development as well.

These two methods were combined to result in a new software development method. In fact, XP has been combined with some elements of Kanban.

How this combination works in practice was tested in summer 2011 within a class at the University of Technology Graz. Nine software development teams with ten team members each tested this software development scheme. A whole, defined development process and the results of its appliance are treated in this thesis.

2. Classical Software Development

The software development process introduced in this thesis is made out of several tools and techniques used in agile software development methodologies. As described above, agile software development is an iterative process and allows rearranging the direction a software project is heading to without much effort. The ability for doing this comes from all participating parties, supporting this kind of behavior.

Nevertheless not every software development team is using agile methodologies.

In the early days of software engineering there was a simple way of developing software called *code and fix*. Someone who starts coding without specification and design, tests and adjusts it until the software behaves like wanted, is using the "model" called *code and fix*. (Ludewig und Lichter 2007).

One could say that code and fix is a fast and lightweight way of developing software. But this behavior comes with a lot of disadvantages as well which, in sum, weight much more than the advantages.

- Work cannot be split into packages to be distributed to more than one developer.
- Requirements are not defined and the result will usually not be satisfactory.
- Quality cannot be measured because defined goals are missing.
- The architecture of a program is bad and its maintenance therefore difficult.
- The knowledge of concepts and decisions is not documented. Only developers know them. This knowledge cannot easily be transferred.

(Ludewig und Lichter 2007)

There are also classical methodologies of software development between *code and fix* and agile development. Nowadays a lot of projects are done a classical, established way of software development.

When building a house, there are not just construction workers laying brick by brick until the house is finished. There must be proper planning, an approval from public authorities and the acceptance of it. The organization of all this must be included. When building software all of this has to be done too to get a good product. Building proper software also may require steps as follows.

- Analysis
- Requirement specification
- First draft
- Detailed draft
- Coding
- Integration and test
- Deployment and maintenance

(Ludewig und Lichter 2007)

Two established methodologies for classical software development are described in the next subsections.

2.1. Waterfall

According to the described steps above, Royce invented the waterfall model. It illustrates the activities of software engineering (Ludewig und Lichter 2007).

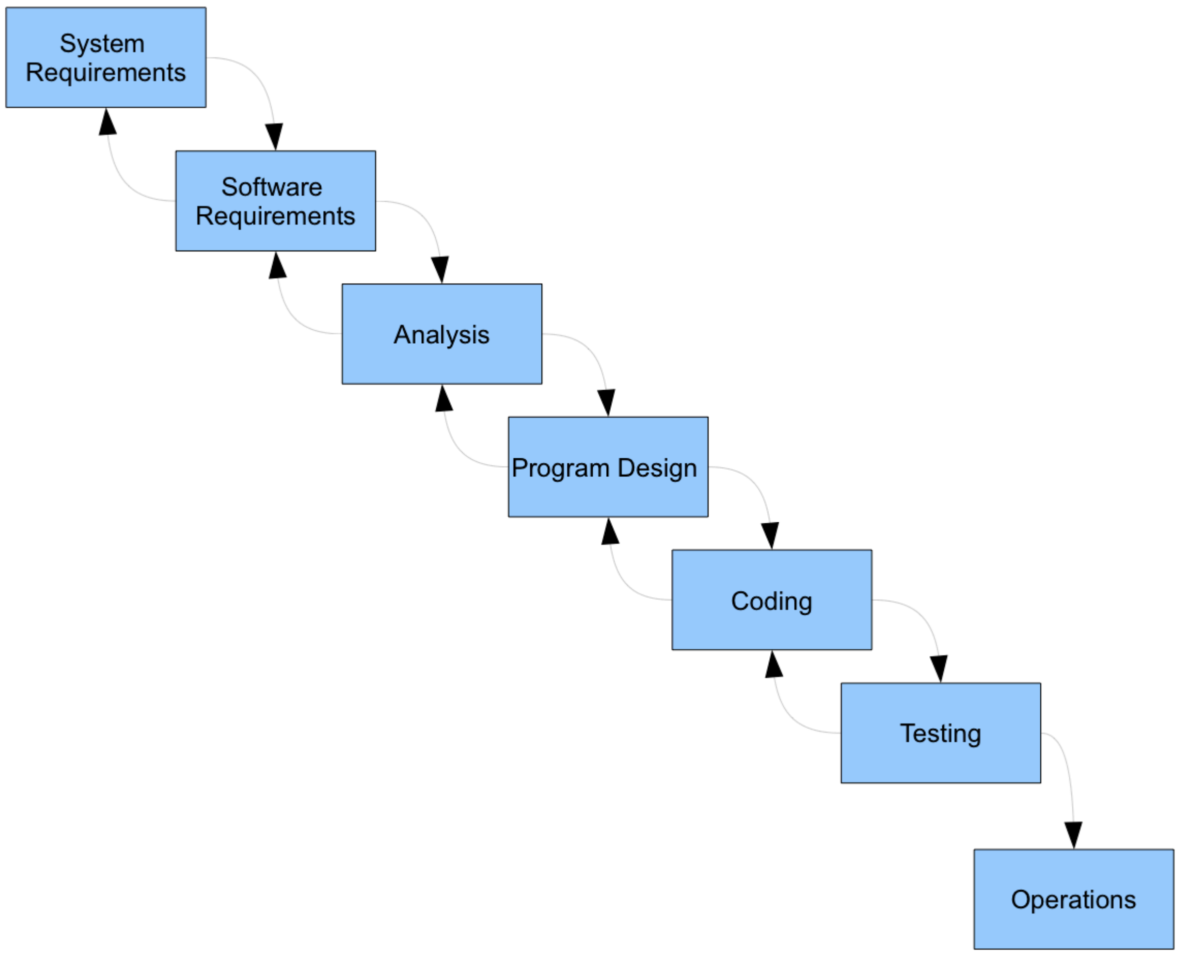


Figure 1 Waterfall model according to Royce

Figure 1 shows a self-drawn model copied from the first official publication of Royce (Royce 1970).

The waterfall model starts with the upper left activity and flows down step by step to the lower right (like a waterfall does). In every step of this model it can occur that errors or mistakes made in the previous step are discovered. The connection to the previous activity allows going back one step and fixing the mistakes made (Ludewig und Lichter 2007).

Royce early addresses a drawback of this model which often occurs in the testing phase. In this phase for the first time the outcome of the software project is measured. If some of the most important external constraints are not met, a major redesign of the software is required (Royce 1970).

To make a redesign, a step back to the design phase or even back to the software requirement phase would be necessary. The problem here is that the model does not support these activities. If something went wrong in the requirements/analysis/design phase, it can take a long time until such mistakes are discovered and it can take even more time to redesign and fix the resulting problems.

One attempt to cope with the problem described above is the development of prototype software, which is a kind of a simulation of the required system. This happens in the design phase and can be seen as another little software project. According to some guidelines, building a prototype should typically take a fourth of the time the whole software project should last. It also involves the design, analysis, design, coding, and testing phase. Royce calls this "do it twice":

> "Without this simulation the project manager is at the mercy if human judgment. With the simulation he can at least perform experimental tests of some key hypotheses and scope down what remains for human judgment …" (Royce 1970).

2.2. V-Model

The waterfall model explained above is a good advisor for project managers and programmers to know what to do and when to do it. Nevertheless, such a procedure model does not say anything about:

- Organization of human resources
- Structure of documentation
- Charge of activities and documents

(Ludewig und Lichter 2007)

Including these points into a procedure model makes a process model out of it. A standardized process model in a company or department of a company enables:

- Providing a reusable planning template
- Introduction of common tools and methods
- Comparison and exchange of results
- Detection of weaknesses and learning from gained know-how

(Ludewig und Lichter 2007)

In literature there can be found a lot of different process models. The V-Model is one of these.

As a further development of the waterfall model, the V-Model is a standard developed in Germany (Wing, Woodcock und Davies 1999). This standard helps customers to keep costs in focus whereas appropriate quality should be guaranteed. So some enterprises make the use of this standard obligatory for all projects they develop with other contractors (Ludewig und Lichter 2007).

Some characteristics of the V-Model are:

- It splits a project into several phases. At the end of each phase there is a milestone.
- The V-Model introduces project-accompanying activities, especially quality insurance, configuration management and project management.
- It is highly expandable and customizable
- Principal and agent are clearly divided roles. The integration of these two roles in the project is highly encouraged.

(Ludewig und Lichter 2007)

2.2.1. Elements

The V-Model consists of defined elements. They can be of the type *activity*, *product* or *role*.

Products are created in *activities*. The newest version of the V-Model, called V-Model XT (XT for extreme tailoring) defines more than 90 activities.

Products are the intermediate results of a project. There are more than 100 products defined.

Roles describe tasks, responsibilities and the required capabilities. There is one responsible role assigned for each product. Additionally there can be more collaborative roles be assigned to a product.

(Ludewig und Lichter 2007)

2.2.2. Conclusion

The short summary of the V-Model above shows that it consists of a high number of defined elements. It is a heavyweight process with a lot of rules. That causes a lot administrative overhead, which only pays off on large software projects with a lot of developers

3. Agile Software Development

"Agile Software Development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams" (Wikipedia, Agile software development 2011).

According to Beyer, agile software development should be an advantage for both, engineers and management. "Breaking the project into short sprints, freezing requirements during the sprint, and getting feedback throughout each sprint are ways of controlling the chaos of software development experienced by engineers". On the other hand, management is convinced by agile development "because it gives them insight and control of a software project. Instead of waiting for months or years, only to discover that the final product is still months or years away, management gets a readout every few weeks" (Beyer 2010).

3.1. Extreme Programming

Extreme programming (XP) is one of the most uncompromising development methodologies. It is "intended to improve software quality and responsiveness to changing customer requirements" (Wikipedia, Extreme Programming 2011).

The "inventor" Kent Beck calls XP "a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software" (Beck, Extreme Programming Explained: Embrace Change 2000).

3.1.1. Values and Principles

XP is based on a number of values and principles. The whole project team has to believe in these values and should follow the principles because they form the basis success with XP.

Values

XP has four base values a team hast to support to make XP work. Without committing to these values, there will be no benefit from just applying XP methodologies (Lippert, Roock und Wolf 2002).

Simplicity

The idea behind this value is to always aim for the simplest solution when developing software. Simple solutions are easy to understand, easy to implement and easy to refactor (Lippert, Roock und Wolf 2002).

Beck explains the simplicity rule this way: "Better do a single thing today and pay a little more tomorrow to change it if it needs it, than to do a more complicated thing today that may never be used anyway" (Beck, Extreme Programming Explained: Embrace Change 2000).

Feedback

According to Beck, "feedback about the current state of the system is absolutely priceless" (Beck, Extreme Programming Explained: Embrace Change 2000).

Beck also says that feedback comes in at different levels. Unit tests give feedback about correctly working program logic. Customers get immediate feedback about the costs after a user-story is estimated. The person in charge for administering the story-cards gives feedback about the progress of a project.

Feedback is tightly coupled with simplicity and communication. If you have concrete feedback, it makes easier to communicate. A simple system is easier to test. If you clearly can tell which tests fail, communication about it will be clear.

(Beck, Extreme Programming Explained: Embrace Change 2000)

Communication

Problems in projects can often be tracked down to bad communication. Invalid project progress information can be tracked down to it. This can happen when a programmer does not tell someone about a critical change in the design or a manager does not ask a programmer the right question to get important information about the progress. Important domain decisions can be made wrong if a customer is not asked the right question (Beck, Extreme Programming Explained: Embrace Change 2000).

All project members should communicate intensively. Informal and personal communication is the best way to exchange information effectively and fast. Ambiguities can be cleared up in no time and with little effort. Good and intensive communication could replace the need for documentation. Nevertheless there could still be reasons for system documentation (Lippert, Roock und Wolf 2002).

Courage

All of the three values above take a lot of courage to be applied. It takes courage make simple design decisions because later on a just too simple design could cause a lot of trouble (Lippert, Roock und Wolf 2002).

Throwing away not well working code in order to make it clear and simple takes courage. Testing three design alternatives one day each and then taking the most promising one takes courage. Telling a co-worker that his code could be written a better way is kind of feedback and needs courage. Reporting a technical problem to the manager is communication and takes courage.

"When combined with communication, simplicity and concrete feedback, courage becomes extremely valuable" (Beck, Extreme Programming Explained: Embrace Change 2000).

Principles

Kent Beck describes twelve principles in XP altogether. They can be derived from the four values. The values are not precise enough to make decisions. One person can see a thing as simple but another one can see the same thing as complex. So the four values have to be divided into more concrete principles. Beck has five fundamental principles:

Rapid feedback

Feedback should be given as quickly as possible. In learning psychology it is a known fact that the time between action and feedback is crucial to learning (Beck, Extreme Programming Explained: Embrace Change 2000).

Assume simplicity

Always treat problems as if they could be solved quite simply. In most cases this is true and so there is plenty of time for problems which cannot be solved in a simple way. Traditionally programmers are told to plan for the future but XP goes the other direction and suggests just to do what is absolutely necessary (Beck, Extreme Programming Explained: Embrace Change 2000).

Incremental change

Every problem should be solved in a series of smallest changes. Incremental change can be found in a lot of ways in XP (Beck, Extreme Programming Explained: Embrace Change 2000).

Embracing change

"The best strategy is the one that preserves the most options while actually solving your most pressing problem" (Beck, Extreme Programming Explained: Embrace Change 2000).

All team members must understand that changes have a positive effect on a project (Lippert, Roock und Wolf 2002).

Quality work

According to Beck, "everybody likes doing a good job". The only possible values for quality are "excellent" and "insanely excellent". If the value is not this high, people do not enjoy work and do not work well. The project is very likely to fail (Beck, Extreme Programming Explained: Embrace Change 2000).

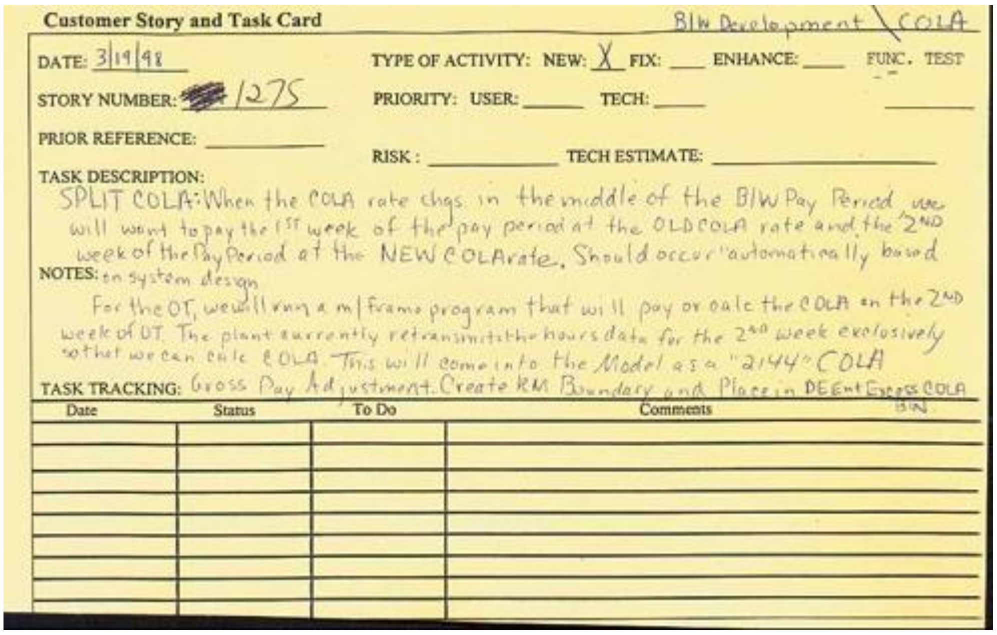
3.1.2. XP Methodologies

Values and principles can be seen as mental tools of XP but there are also practical tools and methods to be used in Extreme Programming. Some of the most characteristic ones for XP are explained here.

Planning Game

This subsection describes how the planning game is done in XP. The planning game is explained in detail here because there are some differences to the planning game in the newly defined process this thesis is about.

According to Beck the goal of the Planning Game is to "maximize the value of software produced by the team" (Beck, Extreme Programming Explained: Embrace Change 2000). It consists of two players, Development and Business and each of them should "invest as little as possible to put the most valuable functionality into production as quickly as possible" (Beck, Extreme Programming Explained: Embrace Change 2000). The Planning Game is a very important part of an Extreme Programming project to get an idea of the features to be implemented and how long it will take to implement them. These features, normally supplied by the customer or the salespeople, are compiled into user stories. A user story is written down onto a story card.



Customer Story and Task Card B/W Development \ COLA

DATE: 3/19/98 TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST

STORY NUMBER: 1275 PRIORITY: USER: ___ TECH: ___ ___

PRIOR REFERENCE: ___ RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION: SPLIT COLA: When the COLA rate chgs in the middle of the B/W Pay Period we will want to pay the 1st week of the pay period at the OLD COLA rate and the 2ND week of the Pay Period at the NEW COLA rate. Should occur "automatically" based on system design

NOTES: For the OT, we will run a m/frame program that will pay or calc the COLA on the 2ND week of OT. The plant currently retransmits the hours data for the 2ND week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in DE Ent Excess COLA BIN

Date	Status	To Do	Comments

Figure 2 A story card (http://xprogramming.com/articles/story_and_task_cards/)

The planning game is a recurring event and consists of three phases where each phase consists of several moves (Beck, Extreme Programming Explained: Embrace Change 2000).

The first phase is the *exploration phase*, in which the team has to figure out what the system should eventually do. Then story cards are written with a description of a feature of the system. This is usually done by Business. After the story is written the Development player estimates how long it will take to implement the story. The Business player supports the Development if anything is unclear. If the story is so complex that it cannot be estimated it has to be split into several stories.

The second phase is called the *commitment phase* where the scope of the next release is set. First of all in this phase Business sorts the story cards by importance (sort by value). Then the stories are sorted by how precisely they can be estimated by Development (sort by risk). Based on this work a set of story cards is chosen by Business (choose scope). This set of story cards has to be implemented for the next release.

In the *steering phase* the project plan is updated according to the things Business and Development have learned from previous iterations. Every one to three weeks there starts a new iteration. An iteration begins with the Business picking the most important stories to be realized. In case the team realizes it has overestimated the velocity for the current release, Business can withdraw the less valuable story cards to fit the new velocity.

If Business needs a new story during development of a release, Development does the estimation of this new story and Business swaps it with one having the same total estimate.

The planning game described above is used for planning a release. Beck also describes the planning game for iterations. This planning game is very similar, "this time though, the pieces are task cards instead of story cards. The players are all the individual programmers. The time scale is shorter. The phases and moves are similar" (Beck, Extreme Programming Explained: Embrace Change 2000).

The first move for a programmer is to take a task card and find a partner. Then test cases are written. When the test cases run without errors, the new code gets integrated and released.

The recovery move is important to not overcharge developers. If a programmer is overcommitted she can ask for help. According to Beck there are five options to recover the programmer:

1) Reduce the scope of some tasks
2) Ask the customer to reduce the scope of some stories
3) Shed nonessential tasks
4) Get more or better help
5) Ask the customer to defer some stories to a later iteration

The final move in the steering phase is to verify an implemented task. Functional tests are run to verify that the story works.

Pair Programming

Pair programming means "all production code is written with two people looking at one machine, with one keyboard and one mouse" (Beck, Extreme Programming Explained: Embrace Change 2000).

One partner is called the "driver". She does the implementation and thinks of the best way to program a method. The other partner takes a more global approach. She thinks about additional test cases or if the whole system can be simplified to make the current problem disappear (Beck, Extreme Programming Explained: Embrace Change 2000).

Laurie Williams says that pair programming is a very good method to make developers learn faster and make them feel more confident. She also argues that joint knowledge can solve a lot of tasks almost immediately (Williams, et al. 2000).

Pairing is a dynamic process. If a programmer is in charge of a task in an area where another developer is more experienced, it is likely that she pairs with such a programmer. The next task may then be done with another partner who needs another developer with expertise (Beck, Extreme Programming Explained: Embrace Change 2000).

Collective Ownership

There are software projects where only the official owner of a piece of code is allowed to change it. If any other team member sees that code needs to be changed, she has to ask the owner of this code to do it. As a result it always takes some time for code to be changed although it should be changed immediately. A second drawback is that just the owner of the code knows it. If the owner leaves the team, all her knowledge about the code is gone as well (Beck, Extreme Programming Explained: Embrace Change 2000).

In XP, everybody is in charge of the whole system. "Not everyone knows every part equally well, although everyone knows something about every part" (Beck, Extreme Programming Explained: Embrace Change 2000). If a programmer pair sees an opportunity to change a piece of code, they should not hesitate to improve it.

Testing

The primary rule on testing is that there must be no feature without an automated test. Programmers write unit tests and customers can write functional tests. Both programmers and customers become confident about the operation of the program. A system becomes very change-friendly this way (Beck, Extreme Programming Explained: Embrace Change 2000).

Refactoring

A programmer in an XP team always thinks of making a system as simple as possible. Even when implementing a new feature, the programmer thinks of ways to add this feature more easily "while still running all tests". This means that sometimes more work is done than necessary to implement a feature. Nevertheless refactoring is done, when the system asks for it, e.g. if there is at first exclusively used code then needed at several positions in a system (Beck, Extreme Programming Explained: Embrace Change 2000).

It is important to have good unit tests to avoid unwanted side effects. So testing is a necessary prerequisite. Bigger changes should be split into smaller ones with just a few minutes of coding each. This minimizes the chance of introducing new errors into the system (Lippert, Roock und Wolf 2002).

40-Hour Week

Kent Beck describes his "perfect" working week as follows:

"I want to be fresh and eager every morning, and tired and satisfied every night. On Friday, I want to be tired and satisfied enough that I feel good about two days to think about something other than work. Then on Monday I want to come in full of fire and ideas." (Beck, Extreme Programming Explained: Embrace Change 2000)

Developers in an XP project should not work longer than 40 hours a week. This way, their creativity and passion are kept high over a long period of time (Lippert, Roock und Wolf 2002).

If there are two weeks in a row with overtime, it is a sign that there is a problem in the project. One week of overtime is fine if the next one can be done in the usual 40 hours, but if overtime is required in the next week too, there must be a problem somewhere in the project's management (Beck, Extreme Programming Explained: Embrace Change 2000).

On-Site Customer

The on-site customer in XP is a person who will be a real user of the system. This way it is guaranteed that a system is built with all requirements implemented since she knows what the system should really be like. It is absolutely necessary that the customer stays with the development team for the whole project. Any questions can be answered immediately and there are no work blocking waiting times.

The customer can also do her normal work while supporting the development team. There just needs to be a contact person in case of a problem is to be clarified (Beck, Extreme Programming Explained: Embrace Change 2000).

3.2. Test Driven Development

The next candidate in the queue of agile software development processes is called *Test-Driven Development* (TDD). The main goal in TDD according to Ron Jeffries is "clean code that works" (Beck 2002). It is an evolutionary approach to software development, which is a combination of test-first programming (TF) and code refactoring (Ambler 2011).

Traditionally a programmer develops code and when it is finished the resulting software is tested. For this purpose, the code is compiled to a running program so the tester can step through it manually checking for proper functionality. If an error in the software is found the programmer is informed and has to change the code which is producing errors. Then the code is again compiled and it is up to the tester to check that the error has gone.

The problem in this procedure is that changing the code could have affected other parts of the software. This can result in new errors (bugs) in parts that were even tested positive before. As a consequence the tester has to test the whole software system every time a change is made to the code. This procedure is quite reactive and can take a lot of time while still it cannot be assured that the tester overlooks errors.

A different approach comes with TF. As the name suggests tests are written before the related piece of production code is developed (Beck 2000).

In addition to TF, the code refactoring procedure is the second part of TDD. Beck defines the mantra of TDD as "Red/green/refactor" (Beck 2002):

1. Red – write a test that does not work (the test suite often signals failing tests with a red "light")
2. Green – make the test work (successful tests are signaled with green)
3. Refactor – remove all duplication created in getting the test to work

Beck also claims that this style dramatically reduces the defect density. Quality assurance can be active instead of reactive now. Unforeseen errors blocking the development process are reduced in a way that management can do better estimation and involve real customers in daily development. Because of the low defect density it is possible to produce "shippable software every day, leading to new business relationships with customers" (Beck 2002).

TF and refactoring are described in the next two sections.

3.2.1. Test-First

As introduced above, the TF approach requires a software developer to write a test case that claims a defined behavior on a specific unit of code. When the test is run it will fail of course because there is no related code yet. After the test is written, the developer writes the code needed to make the test run successfully. In a nutshell, this is all Test-First Development is about.

TF is a practice of XP as well and Beck says that these tests have to be "isolated and automatic" (Beck, Extreme Programming Explained: Embrace Change 2000). An isolated test does not have any connection to other tests. This assures that one failing test does not cause a lot of other tests to fail too. Automatic tests prevent stressed people from being even more stressed by having to manually validate tests. So automatic tests that indicate if the system is behaving well or not are a great help (Beck, Extreme Programming Explained: Embrace Change 2000).

Tests can be seen as another form of communication and documentation. Tests execute classes and methods and so the test code evolves as critical part of the system documentation. Tests also communicate the software design because they show in detail how to execute the class's functionality. Tests are also a safety net because a coding mistake is likely to be detected by a failing test. As a result, people working on the code do not fear refactoring and maintenance activities, which furthermore leads to cleaner and better code (Madeyski 2010).

The workflow of Test-First Programming is shown in the activity diagram in Figure 3.

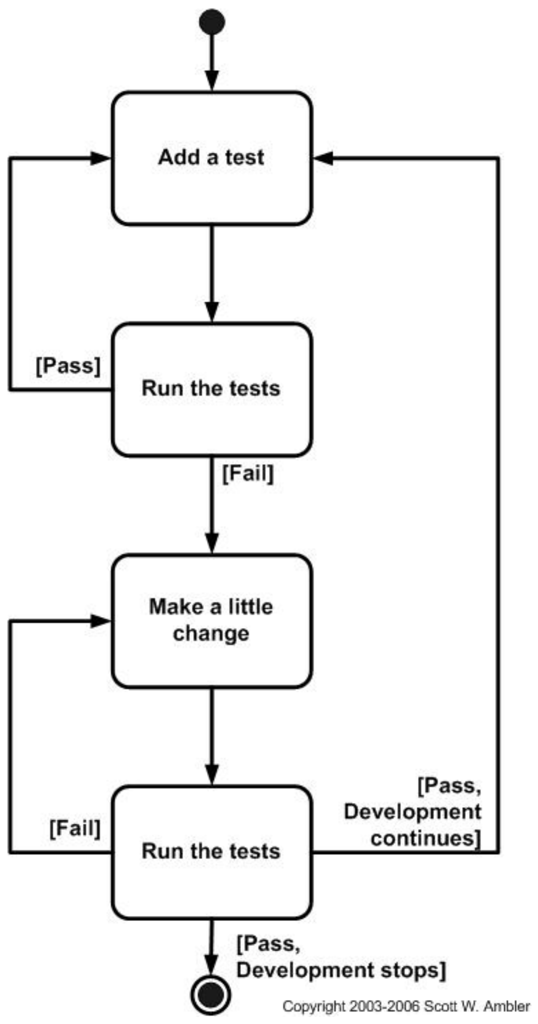

Add a test
Run the tests
[Pass]
[Fail]
Make a little change
Run the tests
[Fail]
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

Figure 3 Test-First activity diagram

The first step is to add a new test. Then run the tests showing that the new test fails. Implement the necessary code to make the new test run successfully. If this is achieved a new test for a new piece of code can be written and the workflow starts again from the beginning (Ambler 2011).

Both, programmers and customers can write tests. Programmers write tests method-by-method. If one of the tests fails, the most important job for the developer is to fix the broken test. This can be done in a minute, but it could also take a month to get rid of the problem. It is very important to fix the test to get back to work as usual as soon as possible (Beck, Extreme Programming Explained: Embrace Change 2000).

The customers write tests story-by-story. They have to define what has to be checked to make sure that the story is done. Every point to be checked is a functional test then (Beck, Extreme Programming Explained: Embrace Change 2000).

3.2.2. Refactor

As mentioned in the introduction of the section on Test Driven Development the process TDD consists of Test-First programming and refactoring. TF builds the base for good refactoring. In a badly tested system the fear of refactoring is big because of the chance of unrecognized negative side effects. Automated tests can give immediate feedback to developers if cleaning up the code did have any side effects or everything still works fine.

Design decisions are made at development time, and in an agile environment there is always a chance of changing requirements. So there is the need for a refactoring-friendly architecture.

Successfully running tests indicate that the system is behaving as it is supposed to. Then it is time to make the code clean. Beck says to first "make it run" and then "make it right" (Beck 2002). Figure 4 from Ambler shows the different states in the TDD process (Ambler 2011).

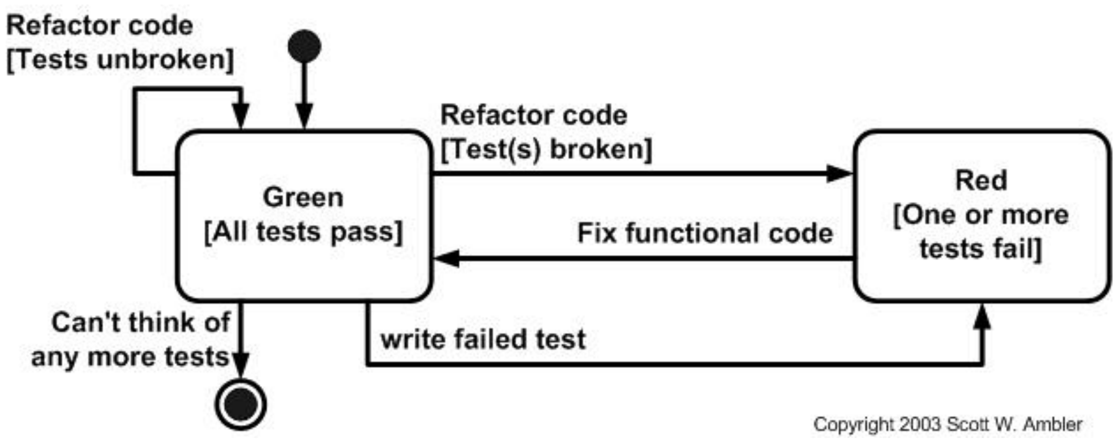

Refactor code [Tests unbroken]
Green [All tests pass]
Refactor code [Test(s) broken]
Red [One or more tests fail]
Fix functional code
Can't think of any more tests
write failed test
Copyright 2003 Scott W. Ambler

Figure 4 The states of TDD

In general, refactoring should be done until one or more tests fail or there is nothing left to be refactored. If tests fail the next step is to fix the functional code that is affected. When the code is clean, every test runs and there are no tests left to think about one is done. More about TDD, refactoring and testing design patterns can be found in Kent Beck's book "Test-Driven Development by Example" (Beck 2002).

3.3. Kanban

A lot of terminology in lean software development comes from Japan and so does Kanban. "Kan means visual, and 'ban' means card or board" (Patton 2009). Initially coming from Toyota as manufacturing industries, Kanban has been adapted to software development as well. Kanban is a system where new work is pulled in on demand but this new work can only be pulled in if there is enough capacity for it in the system (Anderson 2010). In literature the whole development method is called "Kanban". The cards used within the process are called "Kanban cards".

Patton describes the car production process at Toyota as an example for Kanban:

"Picture yourself on a Toyota production line. You put doors on cars. You have a stack of 10 or so doors. As you keep bolting them on, your stack of doors gets shorter. When you get down to 5 doors, sitting on top of the 5th door in the stack is a card — a Kanban card — that says [...] to build exactly 10 more car doors.

You pick the Kanban card up, and run it over to the guy who builds doors. [...] He takes your Kanban card and begins to build doors.

You go back to your workstation, and just a bit before your stack of doors is gone, the door manufacturer comes back with a stack of 10 doors. You know that the Kanban card has to be slid in between doors 5 & 6. You got the doors just in time" (Patton 2009).

A more formal and universal approach comes from Anderson:

"A number of Kanban (or cards) equivalent to the (agreed) capacity of a system are placed in circulation. One card attaches to one piece of work. Each card acts as a signaling mechanism. A new piece of work can be started only when a card is available. When some work is completed, its card is detached and recycled. With a card now free, a new piece of work in the queuing can be started. " (Anderson 2010).

Kanban is not just used in manufacturing industries. As mentioned above it can also be applied in software development. There is a virtual Kanban system in this case and a Kanban card is not a signal card anymore. A Kanban card then can be seen as a work item to be done while developing a piece of software. The signal to pull a new work item occurs when the capacity (or limit) is not reached yet. This signal can be inferred from a physical board using sticky paper sheets or from a software-based work-tracking system. Nevertheless a card wall with “sticky note work items“ like commonly used in agile software development is not immediately a Kanban system. It is just a visual control system that can be used for Kanban (Anderson 2010).

Kanban as an agile methodology has the customer at its focus and concentrates on continuously delivering high quality software using just a few rules. Kanban puts “Individuals and Interactions over Processes and Tools” (K. Beck, B. Mike und v. Arie, et al. 2001). Therefore a crucial feature of Kanban are “Work In Progress”-Limits. These limits are used to set the maximum capacity of the Kanban system and to anticipate bottlenecks in the development process. Not overloading the system prevents the development team members from being stressed, which in turn results in higher quality of code.

3.3.1. Kanban Board

One important part of the core of Kanban in software development is the Kanban board. As mentioned above a card wall can be part of any agile methodology but Kanban gives additional value to it (more on this in the next section).

There is no defined layout of a Kanban board. Every software development team has different resources and needs, resulting in different board layouts. Things all Kanban boards have in common:

- They are divided into columns
- They use Kanban cards
- The most outer left and right columns indicate the start and end points of the Kanban workflow

“The basic idea is stories start on the left side of the board and race across the board through the phases of development they need to go through to be considered ‘done’” (Patton 2009).

Team size and formation influence the look of a Kanban board. Figure 5 shows a very basic example of such a board.

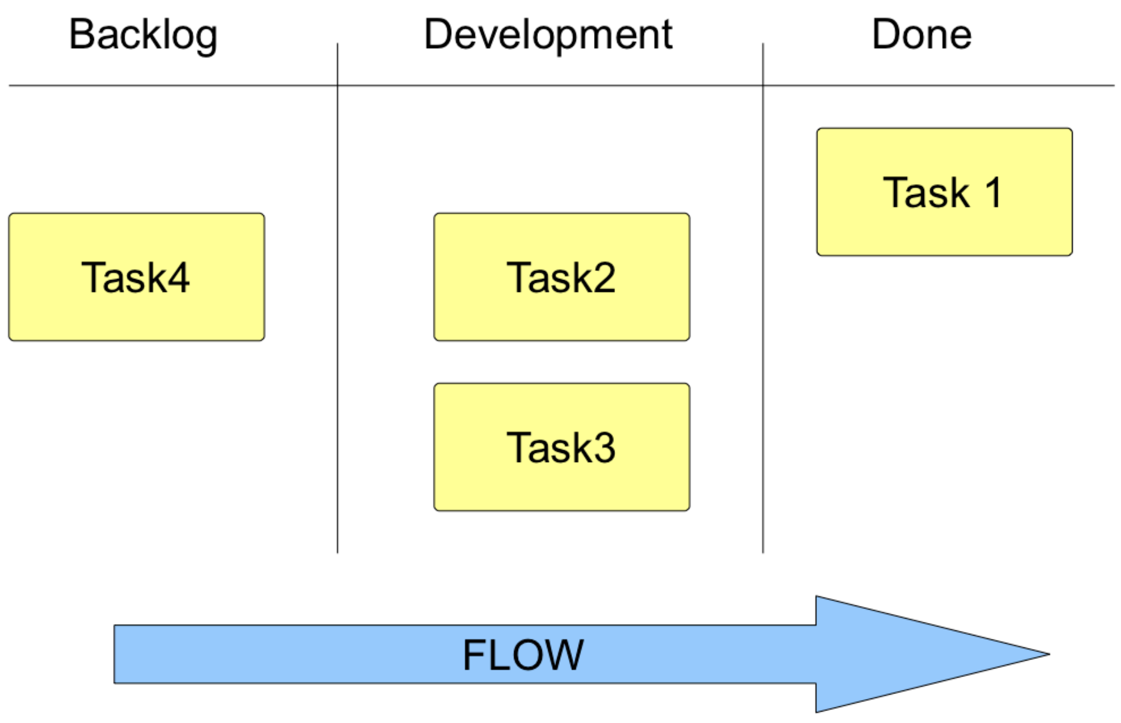


Figure 5 Basic Kanban board

The board above just consists of three columns, where *Backlog* is the start point and *Done* marks the end point of the workflow. Between these two columns there is just a *Development* column indicating the tasks within this column are currently being implemented. There is no rule what such a task must look like. A task can be a user story, a bug report or even a change request. It can be useful to use different colors for different types of tasks.

Since there is just the Development column, all necessary steps like analysis, programming, testing and integration are done within this column. This card wall design would not create much value on workflow or performance transparency. For a better outcome it would be a good change to split the middle column into several ones indicating different actions.

Before drawing an appropriate Kanban board the workflow within a software development team has to be analyzed. When this is done the columns of the board are drawn representing activities of the workflow (Anderson 2010). This customization can maximize the gain a team can achieve from a card wall.

According to the best practice approach above a more sophisticated and valuable Kanban board can look like shown in Figure 6. Henrik and Mattias suggest keeping a Kanban board as simple as possible though (Henrik und Mattias 2010).

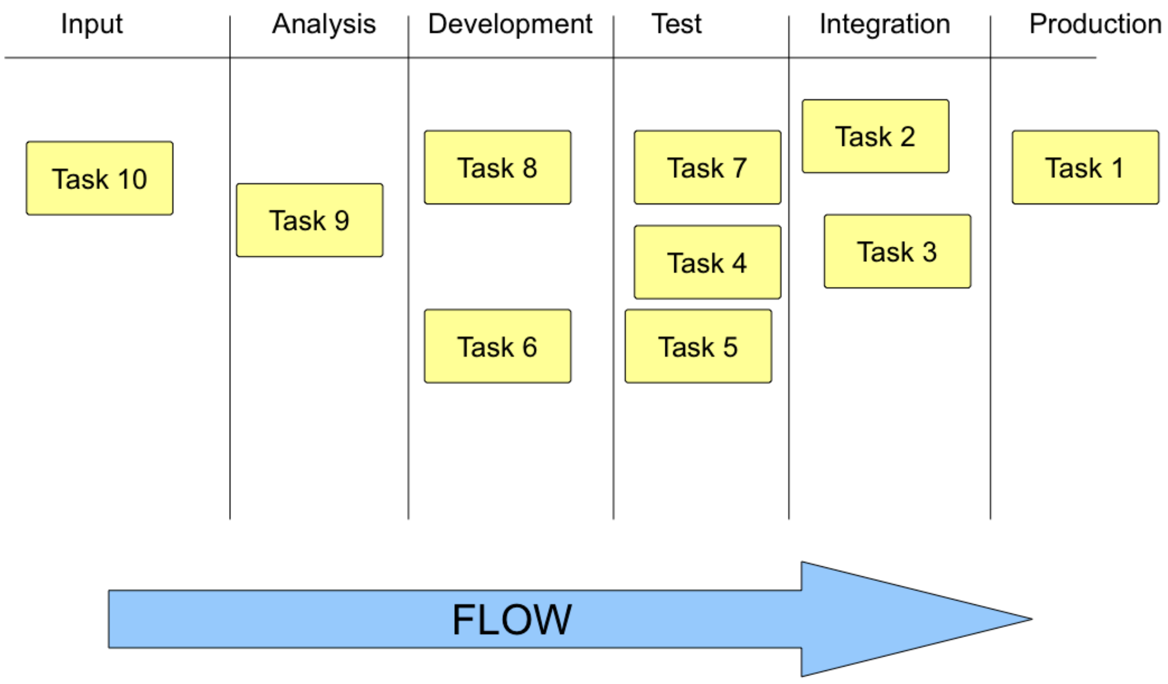


Figure 6 A more complex Kanban board

Again, the starting point is on the leftmost side. This time it is called *Input* instead of *Backlog*. The wording of the columns can be chosen freely of course. The end point is called *Production*, which indicates that in this case "Task 1" is already integrated in the production code. The columns in between show the current state of the tasks to be done. A medium size development team can have specialists, e.g. for programming or testing, so there is not just one team member charged for a whole task. As a result a group of developers may just be working on programming tasks, while there are other specialists for testing. A team leader could be responsible for the analysis of work items in advance.

While such a board layout can be used in any agile methodology a Kanban board has a very important unique feature: The "work in progress limits". The next section describes these limits in detail.

3.3.2. WIP Limits

One important characteristic of Kanban is the agreed capacity for a Kanban system. Transferred to a Kanban system this means the number of story cards allowed to be on the board at the same time. The system capacity is called "work in progress limit" (WIP limit).

A common value for a WIP limit to start with is the number of knowledge workers in the team. Some resources suggest to use a WIP limit half the number of team members (Patton 2009). Others point to research, which suggests a WIP limit twice the number of members (Anderson 2010). In fact, a WIP limit can be empirically adjusted. If the limit selected at the beginning is not working well it can be adjusted up or down.

The overall WIP limit now has to be split up and assigned to the columns on the Kanban board. The number below the column description indicates the actual limit within the column. If there is a limit of "3" on the "Development" column, then no more than three user stories are allowed to be within this column at the same time.

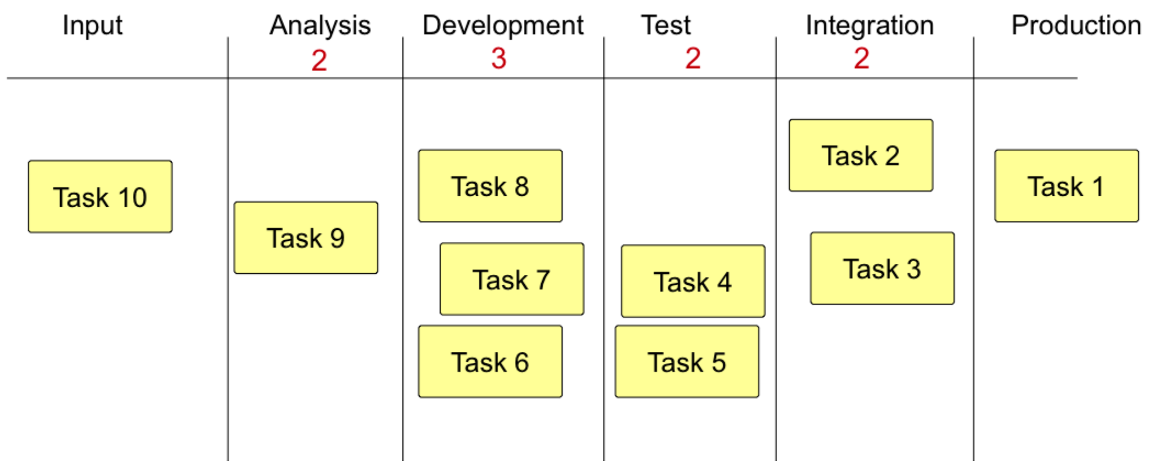


Figure 7 Kanban board with WIP limits

Figure 7 shows a Kanban board with WIP limits in red. The sum of all column limits is equal to the size of the Kanban system. In this case a total of nine story cards are allowed to be in progress. However e.g. just three cards are allowed to be in the "Development" column at the same time. Only when a story is completely developed its corresponding card can be moved to the next column and a new card can be pulled in from the previous column. The goal of limiting the columns is to avoid unwanted "phantom traffic jams" in the development lifecycle.

Atreya uses a traffic jam example to show how important WIP limits are. He draws an analogy between traffic jams and the time it takes to complete a user story. A phantom traffic jam is a traffic jam caused just by variations in driver behavior. There is no need for an accident or lane closure. "Just one driver braking too hard can cause a phantom traffic jam 8 to 10kms behind" (Atreya 2011).

Traffic density can be compared to the backlog list of the Kanban board. And the estimated time of how long a story takes is the pendant to variations in driver behavior. "If your backlog increases at a greater rate than you can deliver, this indicates that the traffic density is increasing and the risk of a phantom traffic jam increases. […] Limiting WIP to match your team's development capacity helps ensure the traffic density does not increase the capacity of your team" (Atreya 2011).

In order to get an idea of how well a development methodology is working one can use some metrics. Above all, numbers are always good for management reporting but also an awareness of the development pace can be a good base for continuous improvement in a team.

3.3.3. Metrics

Kanban is used to produce deliverable software anytime and therefore it is less important to know if the project is "on time". In Kanban as a continuous-flow system it is more important to show the system's predictability and that it is operating as designed (Anderson 2010).

One metric that can easily be measured is the *story cycle time*. A user story is marked with a date when it is created ("entry"), when its development has begun ("started") and when it has been finished ("done"). The difference between the creation time and the completion time includes the waiting time in the backlog until its development begins. The average cycle time from the already completed stories can be used as estimation for when a new story will be completed. The time between "started" and "done" is the *working cycle time* (Patton 2009).

After a time with several completed story cards there will be a relatively stable value for cycle time. This metric is then useful for all team members, stake- and shareholders. It is good to know that a newly created user story takes, say, 10 days to be completed. Both values also include times when an item sits idle between two columns. An average cycle time of two weeks for a trivial user story card can be an indicator that the system is not well-balanced. The WIP should be improved somewhere then (Patton 2009). Anderson calls the metric mentioned above "*lead time"* (Anderson 2010).

Another metric Anderson mentions is *throughput*. "Throughput should be reported as the number of items [...] that were delivered in a given time period, such as one month" (Anderson 2010). Of course the goal is to increase the throughput. In agile fields this is also called "velocity" (Anderson 2010). "Throughput is used as an indicator of how well the system [...] is performing and to demonstrate continuous improvement" (Anderson 2010).

These were just a few metrics, which can easily be created with only using a Kanban system. A lot more are described by Anderson (Anderson 2010).

3.4. Scrum

The last agile software development method to be mentioned in this thesis is called Scrum. Pichler calls it a "management framework for developing software" (Pichler 2008). Unlike a precisely defined process, there is no detailed description on how everything is to be done on the project. Much is left up to the software development team (Cohn 2011).

As an agile framework, Scrum sticks to the values of the agile manifest (K. Beck, B. Mike und v. Arie, et al. 2001). Therefore, Scrum focuses on the human being during software development. There is a special focus on the customer and on the development team, which can act in a self-organized fashion since it knows best how a problem can be solved.

There are just a few artifacts and rules in Scrum. The three defined roles in Scrum are the *development team*, the *scrum master* and a *product owner.* The scrum master can be seen as the coach of the team. The product owner is a customer involved to make sure the product is built for the proper needs (Cohn 2011).

One of the main artifacts of Scrum is called *user story*. It is a short description of a feature the user wants to have implemented in the future product. All user stories are collected in a *product backlog*, which should in sum contain all necessary tasks and stories to build the desired software.

The main activity in Scrum is called *sprint*.

3.4.1. Sprint

Mike Cohn explains that Scrum is built of a set of sprints.

"Scrum is an iterative and incremental process and so the project is split into a series of consecutive sprints" (Cohn 2011).

A sprint can be seen as a time box in which a set of selected user stories have to be completed. During a project there is one sprint after another until the customer is satisfied with the product. Before a sprint begins the set of tasks to be done is selected. This set of user stories is called *sprint backlog*.

The agile aspect of scrum lies between the sprints. The scrum team can edit the product backlog and re-prioritize the user stories. But when the team has agreed on the sprint backlog and the sprint has started, no changes must be made to the sprint backlog.

In the beginning of a scrum project, the length of a sprint needs to be found. The first iterations are used to get a good feeling about the best sprint length. If there is a proper length found it has to be officially defined. The sprint length is fixed and the amount of stories to be implemented is variable since each story has a different level of complexity and thus takes more time ore less.

The advantages of a short sprint time are (Kniberg 2007):

- Short feedback cycle
- More frequent deliveries
- More frequent customer feedback

- Less time spent running in the wrong direction
- Learn and improve faster

The positive aspect of long sprints is that there is less overhead from sprint planning etc., also long sprints allow the developer team more room for recovering from problems within a sprint and still making the sprint goal (Kniberg 2007).

A challenging task in Scrum is to estimate the number of feasible user stories to be chosen for a sprint. This can get better the longer the scrum team is working together. The progress within a sprint is visualized by means of a *burndown chart*. A burndown chart basically shows how much work is left and if the team is on track within a sprint (Cohn 2011).

3.4.2. Comparison to XP

The sections above were just a short introduction to Scrum as a development framework with the most important points of Scrum picked out to make a comparison to XP possible.

First of all Scrum does not have as many rules as XP does. This is reflected by calling Scrum a framework and not a process. Scrum does not make any suggestions on how to improve software quality. There is no pair programming or test-first mentality.

Without pair programming there is no automatic knowledge sharing process, which is a very important feature of XP. Weaker team members do not benefit from a steeper learning curve.

Another difference is that the team is committed to the defined load of work within the sprint. It is not able to act immediately upon changes in the environment. E.g. if there is an important bugfix request coming up during a sprint there is no way to integrate it to the current sprint without breaking the rules of Scrum.

4. The New Approach

The main idea of this work is to combine the previously described XP and Kanban methods. The components introduced in the Kanban method like Kanban card and Kanban board can be seen as a lightweight work package management system. These were fitted to the purpose of use with XP.

The outcome is described in the following sections.

4.1. Planning Game

The planning game is the first thing to be done before starting development and it is the initial step of each iteration loop, too. The on-site customer brings in all user requirements and stories. These new stories have to be formalized and written down onto story cards.

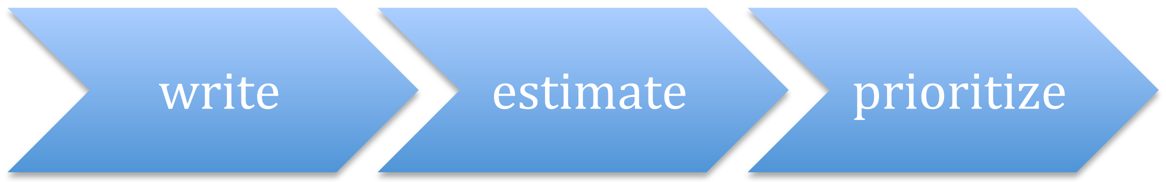


Figure 8 Three steps of the planning game

At first, the on-site customer, supported by the manager and the developers, writes new story cards. Each story card should contain one user story. Support by developers and the manager ensures that every team member understands the story. An advantage of this is the use of a common terminology. Beck recommends to immediately give feedback to the written stories so customers can "learn quickly to specify what the programmers need and not specify what the programmers don't need" (Beck, Extreme Programming Explained: Embrace Change 2000). If customers and developers work together from the beginning this learning effect can be realized quickly.

Customers who are not technical experts could use other words or terms as developers for the same thing. This can lead to misunderstandings when implementing a story card. If all involved members agree on a certain story and its description, the chance of ambiguous wording is minimized.

After all stories are written, developers have to estimate the effort it will take to implement each. The on-site customer supports the developers here, in case anything is still unclear about the story. The estimated value has to be a consensus of all developers. If the estimates to a story differ widely, they have to be discussed among the developers (Sillitti 2010). The discussion and the agreement on an estimated value is also called *planning poker* (Grenning 2002).

In the planning poker, each developer has to bring arguments for his estimate. Some developers may know more than others on a specific topic but developers can be convinced with a short explanation so the value resulting from the most plausible reason is taken. The developer who "won" the planning poker has to be written down on the story card so she can be questioned later on if problems occur when the user-story gets implemented.

The last step of the planning game is to prioritize the stories. The customer now knows the effort for each story to be implemented. Supported by the developers, the customer selects the most important user stories and gives them a priority number. The priority is coded as a single number from 1 to infinity, where 1 means highest priority. The higher the number gets the lower the priority is. The preferred way is to give core features high priority and "nice to have" features higher numbers, which means lower priority. Good practice for creating priority numbers is to assign numbers in steps of ten so user stories created later on can be put in between if it is required.

4.2. Story Card

A story card in this new approach is an adapted version of the story card described in the Extreme Programming (3.1) section. Like the XP story card, it is a high level description from which task cards (XP) or work item cards (Kanban) are generated.

These story cards are designed to fulfill all necessary task-management requirements and to keep administrative effort low.

The next figure shows what such a story card looks like.

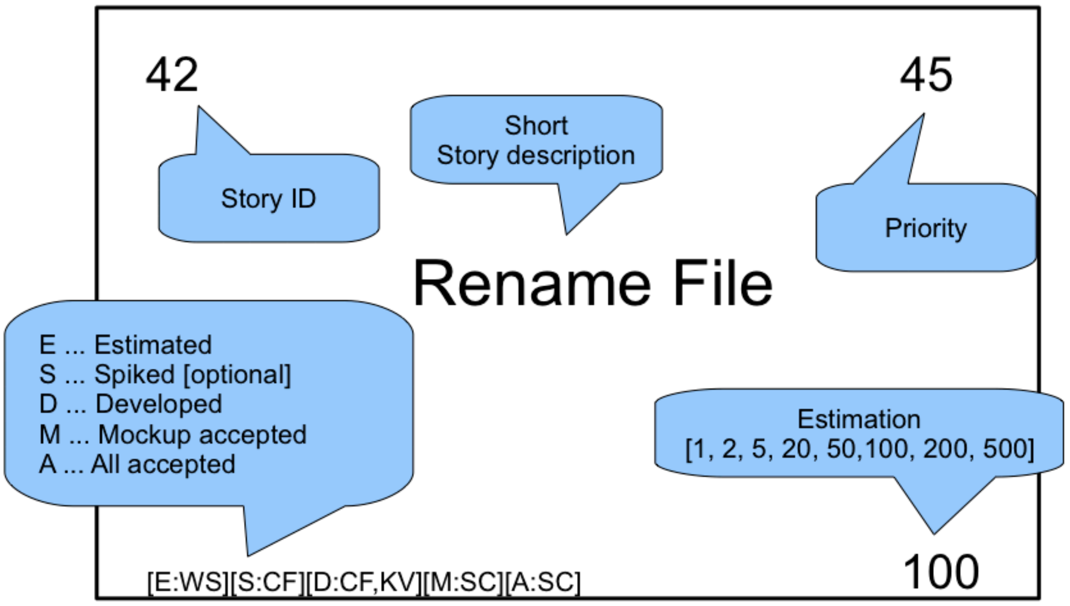


Figure 9 An adapted story card

First of all, a story card has a unique ID. This is placed on the upper left corner of the card. IDs are an incrementing sequence of numbers starting at 1.

In the center of the card is the description of the user story. Depending on the complexity of the user story there can be just a few words or one to three sentences. A good rule is to be as precise as necessary but as brief as possible. Customer and developers should agree on the description.

On the upper right corner is the priority of the user story. The lower the number, the higher the priority of the story card. More on this topic can be found in the section "Planning Game".

The lower right corner contains the implementation effort estimate made by the developers. For this, a non-linear counting system is used. The estimate is not done in work hours or days. There is just a predefined set of available values that can be used to describe the estimated effort. In this case the available values of roulette chips were taken (1, 2, 5, 20, 50, 100, 200, 500). The idea behind this system is to completely decouple "time-thinking" and effort. A story X with an estimated effort of 20 is just more expensive than a story Y with an estimated effort of 5. These two values should not be compared in a mathematical sense. Yes, 5 times 4 equals 20, but in this system it just means that implementing story X costs more effort than implementing story Y and not that X is four times as expensive as Y. The real value of these estimation points clears up after some iterations when especially developers can compare previously done stories with new ones. Then they get a feeling of where to put the story in means of effort.

Another, but less fine grained estimation, could be made by comparing stories to one another and assigning t-shirt sizes: S, M, L and XL (Blankenship, Bussa und Millett 2011).

In the lower left corner there are some cryptic codes written. The number of these tags increases with the progress of implementation of the story card. Each tag has its own meaning. The pattern of these codes is always the same: A single letter indicating the current state of the story card, a colon and then the initials of the responsible person(s) for this state.

The states are:

- E – estimated
- S – spiked (optional)
- D – developing
- M – mockup accepted
- A – all accepted

If Kent Beck did the *estimation* of story card X, then the first tag in the lower left corner would be *[E:KB]*.

If an idle developer does not immediately have another free developer to build a pair with, she can still take the next story card in the priority queue and start *spiking*. Spiking can be seen as a research process done on the topic of the upcoming story card. Spiking itself is explained later on in detail. If someone spiked on a story, it has to be remarked on the story card with a tag: *[S:KB]*. This means Kent Beck is/was spiking on the user story. This is the only tag, which can be put optionally onto the story card.

When there are two free developers ready to build a pair they start developing and put the next tag on the story card: *[D:KB,CF]*.

The first step in the developing process is to create a paper mockup for the user story. If the on-site customer is satisfied with the provided mockup she can accept it. This adds another tag to the story card with the ID code of the customer: *[M:OC]*.

Now the developer pair can start implementing the user story. If they are done the user has to accept the outcome before the "all accepted" tag is put onto the card: *[A:OC].* With this tag, the customer states that she is satisfied with the new feature, which is now ready to be integrated in the software.

4.3. Spiking

According to literature, the meaning of the term "spiking" or "spike" can slightly vary.

Wells recommends creating "spike solutions to figure out answers to tough technical or design problems. A spike solution is a very simple program to explore potential solutions" (Wells 1999).

Another description comes from Mary and Tom Poppendieck. They call the spike a "spanning application" with which you have "driven a nail through the system" (Poppendieck und Poppendieck 2003). This is kind of a "proof of concept" solution by developing a small front-to-end application.

Miller has a very similar approach on spiking the one used in this work. He claims that spiking is not just building a prototype. "In fact it's really a learning experience not a building one" (Miller 2007). But for Miller, spiking is a process where all team members are involved to support "cross-functional communication" (Miller 2007).

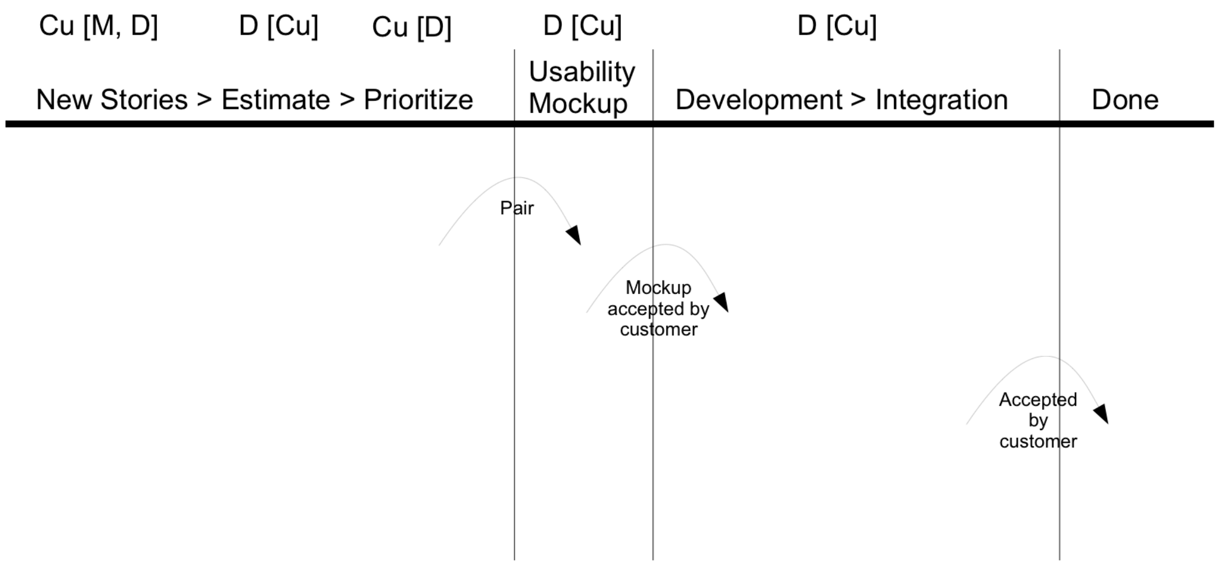
In the context of this thesis, spiking is done when a single developer has no immediate co-worker to build a programmer pair. So she grabs the next story card and starts doing research related to the story. This can be to write some example code to get an idea of how to solve a problem. Another example for spiking is doing Internet research on the selected topic. So, the developer can get comprehensive information that can be used on the user story.

An important note on this topic is that spiking should really be just a learning process and not a process where decisions are made. As a consequence the produced code while spiking should not be used for production. All new productive code should be written with the programming partner. This makes sure that all knowledge is shared.

4.4. Board

The Kanban board had to be customized to meet the requirements of the process described above. This subsection describes the columns used on the Kanban board and why they have been chosen.

Figure 10 shows a sketch of the developed board.


Cu [M, D]
D [Cu]
Cu [D]
D [Cu]
D [Cu]
New Stories > Estimate > Prioritize
Usability Mockup
Development > Integration
Done
Pair
Mockup accepted by customer
Accepted by customer

Cu ...	Customer
M ...	Manager
D ...	Entwickler
[ ] ...	Supporting

Figure 10 the developed Kanban board

The customized board has a reduced number of columns. The XP method uses developer pairs where each pair implements a user story from the beginning to the end. This includes the writing of tests, making mockups, coding and integration. So there is no need for an extra “analysis” or “test” column as described in the Kanban section.

Another important thing is the absence of WIP limits on the columns. WIP limits are obsolete here because XP guarantees that there is no idle developer. Either a developer is working on a story card with another developer or a single developer spikes. The front-to-end development process requires the developer pair to complete a full story. No other developer is involved and there cannot arise a bottleneck on any column.

Story cards on the board are moved from left to right, beginning at the “New Stories > Estimate > Prioritize” column. There are three internal steps done within this column, which are described in the Planning Game (4.1) section.

All story cards are placed on the first column after the planning game is done. This column is also often called backlog. Now, pairing and implementation can begin. The priority defines the next story card to be implemented. Developers build pairs, take the story with the highest priority from the first column and move the card to the next column. The pairing process is explained in detail in the Workflow (4.5) section.

A user story implicates interaction of the user with the system. A user interface has to be designed therefore. The customer must approve this interface. There is no need to program any prototypes to get the acceptance by the customer. A paper mockup is enough for this purpose. A sheet of paper and a pen suffice to draw a simple proposal for the customer. There are two advantages of just sketching on a sheet of paper. Firstly, it is done very fast without much effort. Secondly, there is no barrier on making changes. The old mockup can just been thrown away and a new one is drawn within half a minute. This cannot be done with programmed GUI layouts. If the on-site customer does not accept the interface mockup, a new one must be drawn considering the requirements of the customer. When she accepts the mockup, the customer signs the story card and it can be moved on to the next column: “Development > Integration”.

The customer has a supportive role on the last two steps. The customer can answer questions just in time and there is no need to wait for an email to be answered. Direct and fast communication is a very good way for preventing misunderstandings.

“Development > Integration” is the column where an active story card remains most of the time. The developer pair writes tests, productive code and makes design decisions. There is always a feedback loop with the customer. There also may be some things to clarify because of concurrent story cards other developer pairs work on. If a user story has a close connection to another story the developer pairs must agree on a common interface.

When the coding part is done, the customer must give his acceptance to complete the story card. If she is not satisfied with the outcome the developers must improve the points the customer criticized. This loop lasts until the customer accepts the user story.

The last step is to integrate the developed code in the system. The precise steps for a guaranteed safe code base are explained in the Workflow section.

The story card is finished now and can be moved to the according column on the Kanban board: “Done”.

4.5. Workflow

Now as all used components are described well, the defined workflow for the method has to be introduced. The goal of this workflow is not to restrict the developers in their doing. The goal of the workflow is to make sure that everyone does the right thing at the right time. Sticking to the defined workflow should result in a gain of quality and development speed. A chart of the workflow is shown below.

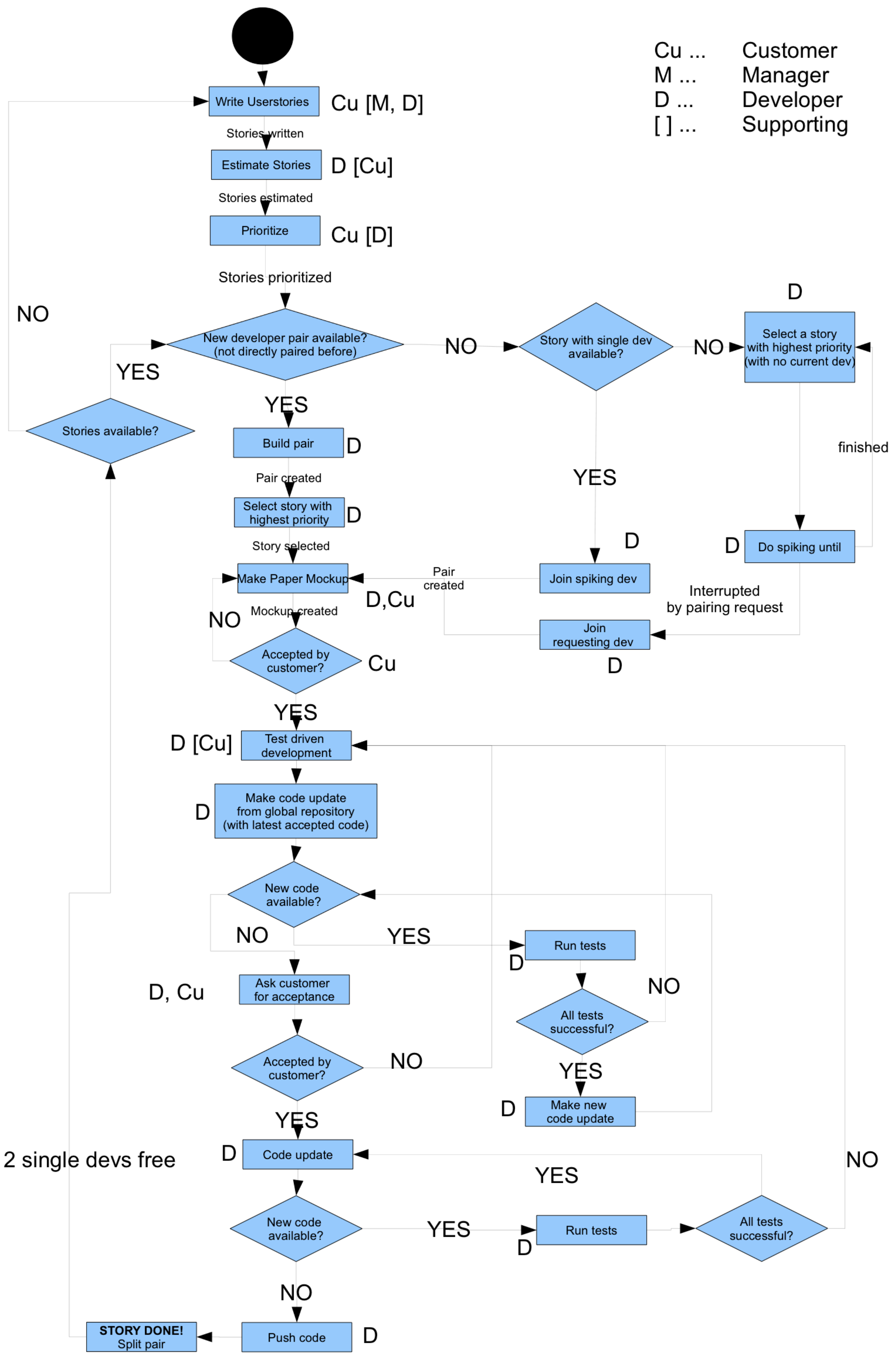


Figure 11 The complete workflow

The workflow begins with writing user stories. This is done primarily by the on-site customer. The team manager and the developers support the customer. The customer tells them what the feature should be like and they find a suitable description of the story all together. This is a good way to make the story understandable to everyone.

The next step is to estimate the effort to implement the user story in the software. The developers of the team do the estimation, since they know best how complicated or how simple it is to implement a specific task. Every developer should feel encouraged to contribute an estimate. Often one of them has more know-how of how the story could be completed or maybe she knows a specific problem that can cause a long completion time. In either case she should tell the others that she knows something special about the story and the estimate is written on the story card. The developer pair which implements the story knows whom to ask then.

The third step when creating a new story card is to prioritize the story. This has to be done by the customer again. After the estimation she knows how "expensive" this story is and she has to decide how important the feature is. A best practice approach is to give priority numbers in steps of e.g. 10. So begin with 10 the 20, 30, and so on. This assures that future story cards can be placed in between already existing ones according to priority. An example: Story A has priority 20 and story B has 30. A new story card X is introduced and it is almost as important as story A. So priority 21 will be assigned which means it will be implemented directly after A was taken.

Once the stories are all written, estimated, and prioritized, the next phase is the implementation phase. Since XP is the development method of choice it has to be ensured that there is always a pair of developers to implement a task. The next step is to build a pair with two developers. In teams with an odd number of developers there is always a single developer left. This single developer has a special job to do. It is called "spiking".

Spiking means that a developer starts a learn or research process for an upcoming task. He takes the story card with the highest priority from the backlog and tries to find as much information as possible to solve this task. This can be doing research through searching the Web, creating mockups, or writing first prototype code. The spiking process has to be considered as learn- and playground of the developer. She is allowed to just "play" and does not have to make decisions during this process. She does not have to find a proper solution for this story. In fact, everything done during spiking must even be thrown out of the code base (see section Spiking).

Spiking is done until there is nothing left to learn for a given task or there is a pairing-request from another developer. In the first case the developer has to take the story-card with the next highest priority and spikes on the new task. In the latter case the two developers build a pair and start implementing the task on which has been spiked before. Now the developer who has been spiking can bring in the experience she gained, which helps to find a good solution for the task.

When a pair is built and a story has been taken, the next step is to create a paper mockup. The customer supports the newly created pair. The customer can bring in his own ideas on the task, and they find a solution together. When the customer accepts the outcome, the developers can go on to start implementing the task. Otherwise the mockup has to be edited until the customer is satisfied.

When the customer has accepted the mockup, the programmers can start implementing the story. Here they use the technique of pair programming and test driven development. During this process the developers can always ask the on-site customer when there is anything unclear about the story. When the story is implemented the result has to be integrated. In a development team of 10 people it can happen that different pairs edit some piece code simultaneously, e.g., when refactoring previously existing code that needs to be changed for several stories on which different pairs are working at that moment. This can lead to merging conflicts on the code base. So before asking the customer for acceptance the developers have to check if there is any new code available on the central code repository. The pair has to merge all new code into their code and run the available tests to check if something does not work anymore, i.e., tests are failing because of the code update. If the changes did break something, the pair has to go back to development and fix the problem. If all tests run successfully, a new code update has to be made. It can happen that in the meantime another pair has committed new code to the code base. If this happens, the procedure has to be repeated until all tests are OK and no new code was committed to the main repository in the mean time. However, even if nothing changed, the code cannot yet be pushed into the main repository.

This is the moment when the customer has to be asked for his acceptance. The developers show him or her what they have implemented. If the customer is not satisfied the developers have to go back to the TDD block in the process chart and they start from there again. If everything is alright there is another "code-update, test" loop to be done. This should avoid that non-working code is committed to the code repository. Only fully tested and working code is committed to the central code repository. A user-story is done when the customer accepted the work, there is no code to be updated, all tests run successfully, and the new code was committed to the central repository into the main branch.

This is the time when the pair splits up and every member gets a new partner. The new pair takes the next story card and a new implementation process starts.

5. Field Study

The whole software development process described above has been introduced in a university course called "Software Engineering and Knowledge Management". Wolfgang Slany held it at the Graz University of Technology in the summer term of 2011. The goal was to test how well the process can be applied to software development teams. In the following subsections, the class and the projects for testing the process are explained.

5.1. Software Engineering and Knowledge Management

In the class "Software Engineering and Knowledge Management" (SW), students get grouped in software development teams of approximately 10 members each. These teams have to build a specified type of software with a special focus on knowledge management. The class is meant to simulate a "real world" project and environment. It involves the whole product lifecycle and includes all phases like planning, design and development. Each team has a designated "customer" as member from whom the requirements for the new software are gathered. The customer is always available to the team and has an active role in the development process. The use of agile software development methods like XP and TDD is mandatory.

Like in real projects there can always be a loss of a team member or the change of the requirements of the software to be built. In the first case, knowledge management comes in place. In the latter case, the use of agile software development is very important. Any time during the term the professor decides to make changes in the teams. The teams never know when such a change will happen so they really have to distribute knowledge well, and they have to choose a software design which enables easy refactoring to meet new requirements. This makes the students aware of the importance of effective knowledge management and efficient software development methods in practice (based on Slany 2011).

In the term of summer 2011, 90 students attended SW. Nine software development teams were built. The team-members were selected according to a previously executed survey where the students were asked about their technical and soft skills. Based on the results of this survey, roles were assigned to the students and well-balanced teams were built then. Available roles according to XP were:

- Customer
- Manager
- Coach
- Developer

The teams were simulating normal working days in a software development company. Every Wednesday they had an eight hours working day with fixed working times and a fixed location, which served as their "office".

5.2. Projects

There were two types of projects defined at the beginning of the term. The goals of these projects were clearly defined. For one of the projects it was up to the team to choose a programming language and technology. In the second one even the programming language was defined, which was *python*. Since there were just a few python programmers available among all the students (known from the survey) there was only one python team built. All other teams were assigned to do the projects of the first type.

The students with the customer role were introduced to the main requirements of the systems to be built. This, combined with the ability to bring in own ideas, ensured that the students were able to act as real on-site customer within their teams.

In the subsections the projects will be explained. The two project-ideas are to be credited to Karl Voit, MSc.

5.2.1. Latex Templates

The first project had to be done by eight development teams. The goal of this project was to develop a web based application for helping users to fill out pre-build templates such as letters written in *LaTeX*[1]. These templates have certain placeholders positioned for

[1] http://www.latex-project.org/

which the web interface should ask a user with what it should be substituted. For a letter this could be "receiver", "subject" or "body". Depending on the template, the system to be developed should ask the user for all the information needed and fill it with the information provided. Then the user gets a download with the previously entered data in the right place so she does not have to edit the LaTeX file by herself.

One important constraint was that this template system should work with not just LaTeX files. It was pretended to be able to use every type of ASCii coded text file using the same placeholder pattern.

Big Change

The big change request in the middle of development was to support not only ASCii files, but also "*Microsoft Office docx*" files. Because of the new requirement there were a lot of changes necessary in the software. This was a good test-piece for the flexibility of the software design.

5.2.2. Tagvizor

The tagvizor project was the one the python team had to develop. Tagvizor is a project based on the PhD research-project "tagstore" by Karl Voit (Voit 2011).

"The main purpose of tagstore is the refinement of a better method to manage files and folders on the local hard disk drive. [...] In a tagstore, files and folders are being tagged by the user. Using those tags, tagstore automatically generates navigation hierarchies called TagTrees. Within those TagTrees the user is able to navigate using his or her preferred software tools like file browsers, 'file save...' or 'file open...' dialogs." (Voit 2011).

The new software called tagvizor should be a browser, providing *faceted search*[1] in stores created by tagstore. Because tagstore is implemented in python, tagvizor had to be implemented in python as well although tagvizor should not be dependent on tagstore.

[1] http://en.wikipedia.org/wiki/Faceted_search

Big Change

The big change request in this software project was to integrate and support "*Google docs*" in the faceted search.

5.3. Pre-Project Survey

In the first few lessons in SW class, the development process and all the components were introduced to the students: The planning game, the Kanban board, the story cards and the overall workflow. There was also a training task where the students had to do the planning game, "development" and refactoring with a number of test questions. The questions were written onto a story card and an estimate of the difficulty was made (planning game). Then these questions had to be answered by "developer-pairs". During the refactoring process the story card had to be passed to another pair, which had to add useful information to the answer.

Now with a basic understanding of the development process the students were asked to fill out an online survey. This survey was designed to ask the students about their feelings for the usefulness and practicability of the method. It was important that the students did not make any practical experiences before the survey was executed. So an unaffected opinion of the students could be captured.

This survey was divided into four main groups:

- general topics,
- Kanban,
- XP and TDD,
- the workflow.

Each group had a few questions, which resulted in a total of 16 questions answered by 75 students. In this section, the results of the survey are summarized. The results in detail are presented in the appendix of the thesis.

Remarking that all questioned people were university students, 72% never developed software in a company. On the one hand this can be seen as a good starting parameter because these students are unprejudiced against any software development process. On the other hand, this reduces the ability to give differentiating answers.

The 28% who were developing software in a company before participated in 7 software projects with 4 members on average. The answers to the question about the development processes used in these projects were ranging from "no process" or "chaotic" over "waterfall model" to agile methods like Scrum. Things that went wrong in the projects were classical ones. No documentation, no communication, no testing, no knowledge management and just wrong effort estimations. The introduced development process is designed to cope with all these points.

66% of the students had no experience with TDD and 72% had no experience with XP before.

More than three quarters of the students declared to understand the rules to be applied to the Kanban board and to the story cards and more than half of the students said that the rules are easy to understand and the Kanban board is administratively valuable.

TDD has not such a positive response in the survey. Less than a half stated that they do not believe that TDD can replace the documentation of code but more than a half believes that they can follow the rules of TDD.

The defined workflow is understandable for more than 90% of the students and approx. 65% think that they can really stick to the workflow.

In conclusion, the bigger part of participants did not have job-related experiences in software development and had a positive attitude towards the software development process introduced to them.

5.4. Intermediate Results

During the semester the author visited the teams. These visits helped to get a feeling about how they are doing with the development process. The manager, the coach and the customer were asked a couple of questions. These questions were about the project progress, the process itself and about problems the team has to cope with.

To get more insight of how the process is applicable, the author himself became customer for two days. This was a good way to point out some difficulties a team can have. This was the base for the questions the other teams were asked.

5.4.1. Identified Problems

During a few days as member of a development team, there came up a number of issues, which are worthy to be addressed.

Unclear user-stories

One problem came with the fact that the projects had an artificial background. The on-site customer was told which product he had to enforce. It was not the idea of the customer herself. This resulted in *unclearly defined user-stories* because she did not exactly know what the requirements are.

Nevertheless, this can occur in real-world projects as well. Customers are often not sure how exactly the software to be developed should behave or has to do.

The customer came with already written story cards to the initial meeting where the first planning game was done. This resulted in a lot of misunderstandings according the used vocabulary. The customer meant completely different things with a word than the developers understood. This made a lot of discussion necessary to get a common understanding of what a user-story should be about.

The quintessence of this problem is that the whole team should create story cards together. The customer just explains his requirements and with the developers' support there are story cards with proper descriptions generated then.

Estimation of unordered story cards

In the first planning game there were a lot of new story cards, which had to be estimated. Proper effort estimation per story card was hard to do because the order in which the stories were estimated was without giving attention to the technical prerequisites.

When estimating a user story, everything to fulfill this requirement has to be taken into account. This includes all infrastructural concerns as well. Here is a concrete example:

The description of a user story says: "User must log in". Unfortunately this is the first story card to be estimated in the planning game. So the developers have to think about how the system to be developed knows about the user who wants to log in. Is there a database with defined users, which are allowed to log in? If not, when, where and how is such a user stored? The estimate here would be high because implementing this story card with all the required background structure can take a long time.

Later during the planning game there is a story card to be estimated saying: "Users can register". The technical background to register a user is the same as stated in the last story card. There must be some storage to persist the user. Chronologically, a registered user is a prerequisite for enabling him to log in. As a consequence the "register" story card should be estimated before the "login" story card.

This example generates the idea to sort the story cards by use-case before to give an estimate.

Duplicate or intersecting story cards

The customer creates story cards with different descriptions meaning almost the same. This is also an outcome of story cards written by the customer herself. Developers soon find out that two story cards are technically very similar to implement. After explaining this to the customer one of the duplicates can be eliminated. If there is a big intersection of two user stories, one of them can be thrown away. The part, which does not intersect, is added to the user story that is kept.

Mockups

Developers are encouraged to draw only paper mockups of the user interface to be implemented for a user story. A sheet of paper and a pencil is enough for this purpose. If the user accepts it, the mockup can be seen as an artifact. The customer and the developers commit themselves to implement the drawn user interface.

A problem is that because of the "unprofessional" look of the self-drawn mockup they are often thrown away after the story card is implemented. The customer cannot verify if the user story is implemented properly.

A mockup must be supplied with the according user-story-id and is part of a story card. During development, the mockup stays with the developer team and afterwards the mockup is stored with the story card. Every decision is documented and can be tracked then.

Blocking story cards

User story B is dependent on the outcome of user story A. Given the case, the dependent user story is next prior to its logical predecessor A. Developer pair X starts with implementing user story A and almost simultaneously developer pair Y starts implementing story B. Pair Y must wait until story A is finished.

5.4.2. Team Interview-Summary

The questions in the team interviews were partly based on the identified problems described above. This section shows the interview questions and a summary of the answers the various teams made.

What were the biggest problems at the planning game?

The teams needed time to get into the planning game. Everything was new, even the applied technology. So an appropriate estimation was hard to do. Often the customer herself did not know what the requirements were, so there was a lot to discuss.

Did you have intersecting story cards? If yes, how did you handle this?

Not clearly defined story cards often lead to intersections. Sometimes the story cards needed to be rewritten and newly estimated. Others became obsolete and were dumped. Some teams had well written story cards so they did not have any intersections.

How did you react when new story cards were required?

New story cards were estimated and prioritized immediately or at least at the next standup-meeting.

Does spiking produce useful output in your team?

Some teams were not familiar with the used technology so instead of spiking, they joined another pair for faster learning. In other teams there was no need to spike because it was always possible to build programmer pairs.

How was the basic GUI implemented? Did you make an explicit story card?

Some teams had explicit story cards for building a basic user interface. Others had a story card where the creation of the user interface was included implicitly. The estimate of the latter one included the GUI creation then.

How do you handle dependencies between story cards?

Basically all teams used direct communication between the affected developer pairs to agree on a common interface. How this method worked was dependent on the programming skills of the developers. If the skills were not high enough, one team had to wait until the preceding user story was implemented.

Do you use automated GUI testing?

Almost all teams could manage to make *Selenium* work for automated web interface tests. The team working on the tagvizor project used a special tool called *sekuli*.

Which versioning system do you use?

The bigger part of teams used *GIT,* and the rest used *mercurial*. Most of the teams had no experience with the usage of versioning systems. Because of that, there occurred a lot of problems. The teams did not apply best practices and went into a lot of trouble when committing code to a repository.

5.4.3. General Remarks

Only few of the students were experienced programmers. Most of the students were not familiar with the technology their team selected and also a lot of them never used a versioning system before. As a consequence the teams had to learn a lot of new things and they could not concentrate on sticking to the workflow. As the team interviews were made in the middle of the term there was high improvement of their skills to be expected.

5.5. Post-Project Survey

This section gives a summary of the results of the survey executed after the project period.

The first intention was to find out how the opinions of the students had changed about the deployed development process during the project. The second intention was to get proper feedback about the process and to get some ideas for improving the process.

Iteration planning: 53% percent of the students think that after some time, it can be precisely estimated how much features can be integrated within an iteration. Some who do not think so gave a comment that it is at least more accurate than with most other techniques.

The planning game is accepted very well to get an overview of the project. 60% percent rated the usefulness with the highest note, 26% with the second highest. 80% of the students think that it is absolutely necessary that every team-member is part of the planning game. One point of criticism was that the bigger the team, the higher the chance of loss of attention.

90% confirm the administrative value of the kanban board. According to the survey answers, the kanban board is easy to read and very valuable in visualizing the project state.

Also 90% say that the rules for the story cards are easy to understand and that they really help in the development process.

Just 40% appreciate spiking as a research technique but it is seen as very useful when there is a lack of knowledge in a particular field of technology. Some answers told that it really helps with developing software faster.

60% did not really make use of paper mockups within their teams. Some of them experienced cases where a story card did not lead to a paper mockup. So the story card was discussed with the customer and a precise textual description was written instead. Nevertheless, a mockup can always be drawn when the user story is written properly.

60% of the students had problems with the defined workflow. The most problems occurred when using a versioning system to merge code branches. This was caused by the lack of knowledge on versioning systems.

89% felt a gain of quality and 78% felt a gain of velocity compared to other projects they participated in before. 100% rate the knowledge-management as very well and would use the introduced process again in other projects.

As a last statement some students remarked that it is very important to note that poorly motivated team members can decrease the efficacy of the team.

5.6. Post-Project Experiences

At the end of the summer term when all teams had reached the deadline for developing their product, the team managers had to present their experiences with the new development process. These reports were called (with a little sense of humor) “war stories”. This included their feelings about XP with all of its methods. They reported about their use of story cards and last but not least about knowledge management. This section gives a summary of these reports concluding the positive and also the negative aspects.

5.6.1. Planning Game

The introduced software development process relies on using user stories as a base for story cards. The fact that these are not split into engineering tasks made the beginning of the projects hard to coordinate. The full team had to start with the project at the same time and there was no project infrastructure available. The whole development environment had to be set up. This would be a classical engineering task, but had to be done within a user story. Without any environment, the programmer pairs were not able to begin with a story card, so they had to wait until the setup was finished. Nevertheless, after some time, the teams were on the right track.

The standup meetings gave a good overview of the current project status. The planning game let the team members know what to do in the project. It encouraged them and made them all to equal parts in the group.

5.6.2. Extreme Programming

There was very good feedback for the methodologies of XP. Pair programming guaranteed a very high level of knowledge sharing. Even the loss of the best developer in the team was not a momentous incident. New team members were integrated very well. Weak programmers could benefit from pair programming. Product quality increased although the pair programming technique did not always support development pace. One team even called XP a "very social way" of developing software. It was also remarked that pair programming could be seen as "immediate code reviews".

The ability for easily implementing change requests was remarked as very positive.

Some teams suggested that pairs should be allowed running two computers: one for development and one for doing research on the current topic, if necessary. Wolfgang Slany noted, however, that this would distract the shotgun partner, thus lowering the continuous code review and knowledge sharing effects.

5.6.3. Test Driven Development

TDD was often seen as very time consuming and developers thought it is too much overhead for small tasks. Especially the writing of automated GUI tests takes a lot of time. Nevertheless the quality and stability of a product resulting from TDD was very appreciated. After a time of acclimatization, TDD was very welcome. It helped with the understanding of code and made further documentation unnecessary. Even not successfully running tests were taken as a positive sign, and encouraged team members to more commitment.

6. Conclusion and Improvements

This section deals with the conclusion of the field study about the combination of Extreme Programming, Test Driven Development and Kanban. It combines personal experience, the result of team interviews, the presented war stories and the survey results. Out of this conclusion, there is a set of constraints generated to improve the efficacy of the software development process.

First of all it is to say that a very big part of the students accepted the process. They were willing to follow all the rules to execute a proper XP project. Novice programmers without any experience with software development processes and also experienced software developers in the teams were really excited about doing software development an "unusual" way.

Speaking of novices, the class in which the field study was made is held in the fourth semester. So students often just have basic experience with one certain programming language. The used technologies in the projects were versatile and often new to them. The project beginnings were difficult because the necessary knowledge had to be gained at first. This included using versioning systems, configuring web-servers, learning a new programming language and not least using the new development process. The students subordinated the focus on the process at first. A team with just experienced software developers, using well-known technologies would be a more efficient, especially in the beginning of a project.

One positive facet of these circumstances was a remarkable high learning curve a lot of students had. This was mentioned by all of the team leaders in their project reports.

Other start problems, like building a proper development infrastructure do not occur in a "natural" project environment. Normally, projects start with an exploration phase and let the developers get a feeling for the technology of choice (Beck, Extreme Programming Explained: Embrace Change 2000, 101). This can be done with a small team where also the infrastructure is developed by a few developers in this phase. After a time, when the base structure is settled and the requirements grow, also the team size can grow.

After solving the start problems, the teams made good progress in developing the required software. From week to week the teams got more used to the whole procedure. Team interviews showed that the teams would have been even more efficient if there was more time available. Project velocity increased from week to week.

One outstanding team in this study was the "python" team. All of its team members had experience in developing in python and so they could be productive just from the beginning. This team had a lot of experts and so it nearly could be compared with a professional software development team.

Some other teams had to cope with "black sheep": Students who did not want to practice pair programming or refused to do Test Driven Development. Those were people who were not used to work in teams and did not want to participate. Normally, in business, such a team member would not be accepted in a development team using extreme programming. The fact that these students could not be thrown out of the class was not a bad thing. The other students saw a good point for how important it is that all team-members show their own commitment to a project. Then, and only then a development process like the used one can succeed.

Other irregularities the teams had to cope with were also from environmental nature. None of the teams had a real office to work in. The greatest part of them worked somewhere at the campus of Graz University of Technology. Beck writes that an optimal environment is absolutely necessary for making extreme programming work (Beck, Extreme Programming Explained: Embrace Change 2000, 119). The lack of fixed rooms also does not allow installing a Kanban Board.

To enable perfect process flow, the environment must be perfect too and no impediments should disturb the development team. Only then they can concentrate on their work.

Even if the conditions for the students were not perfect, they learned how "different" software development can be and they also learned how a single non-committed co-worker can disturb a whole process. They got to know about the importance to be able to react in an agile way to change requests and they also learned about the importance of writing well testable and clean code. All the experience they made is fixed in their memories and changed their thinking about software development. The job of the class is done, even if just a few of the students will use the learned techniques in their future jobs, and if they maybe “infect” others with a highly efficient method to develop software.

7. Literaturverzeichnis

Wells, Don. *ExtremeProgramming.* 1999. http://www.extremeprogramming.org/rules/spike.html (Zugriff am 25. 08 2011).

Wikipedia. *Agile software development.* 26. 07 2011. http://en.wikipedia.org/wiki/Agile_software_development (Zugriff am 01. 08 2011).

—. *Extreme Programming.* 03. 07 2011. http://en.wikipedia.org/wiki/Extreme_Programming (Zugriff am 01. 08 2011).

Williams, Laurie, Robert Kessler, Ward Cunningham, und Ron Jeffries. *Strengthening the Case for Pair-Programming.* Paper, Computer Science, University of Utah, IEEE Software, 2000.

Wing, Jeannette M., Jim Woodcock, und Jim Davies. *FM'99 - Formal Methods.* Berlin: Springer Verlag, 1999.

Voit, Karl. *tagstore.* 01. 08 2011. http://www.tagstore.org (Zugriff am 01. 08 2011).

Abram, Alain. *Guide to the software engineering body of knowledge.* Michigan: IEEE Computer Society, 2004.

Anderson, David J. *Kanban - Successful Evolutionary Change for Your Technology Business.* Sequim, WA: Blue Hole Press, 2010.

Ambler, Scott W. *Introduction to Test Driven Design.* 2011. http://www.agiledata.org/essays/tdd.html (Zugriff am 09. 10 2011).

Atreya, Charan. *Importance of Kanban work-in-progress (WIP) limits.* 16. 01 2011. http://www.atlanticcanadabusinessblog.com/index.php/2011/01/16/project-management/importance-of-kanban-work-in-progress-wip-limits/ (Zugriff am 06. 10 2011).

Beyer, Hugh. *User-Centered Agile Methods.* Morgan & Claypool, 2010.

Beck, Kent. *Extreme Programming Explained: Embrace Change.* Indianpolis: Addison-Wesley, 2000.

—. *Test-Driven Development By Example .* Kent Beck Draft, 2002.

Beck, Kent, et al. *Manifesto for Agile Software Development.* 2001. http://agilemanifesto.org/ (Zugriff am 16. 09 2011).

—. *The Principles Behind the Agile Manifesto.* http://agilemanifesto.org/principles.html (Zugriff am 06. 10 2011).

Blankenship, Jerell, Matthew Bussa, und Scott Millett. *Pro Agile .net Development With Scrum.* Apress, 2011.

Cohn, Mike. *Mountain Goat Software - Introduction to Scrum - An Agile Process.* 2011. http://www.mountaingoatsoftware.com/topics/scrum (Zugriff am 19. 09 2011).

Grenning, James W. *Planning Poker or How to avoid analysis paralysis while release planning.* http://renaissancesoftware.net/, 2002.

Henrik, Kniberg, und Skarin Mattias. *Kanban and Scrum - making the most of both.* C4Media Inc., 2010.

Kniberg, Henrik. *Scrum and XP from the Trenches - How we do Scrum.* C4Media Inc, 2007.

Ludewig, Jochen, und Horst Lichter. *Software Engineering.* Heidelberg: dpunkt.verlag, 2007.

LaTeX – A document preparation system. 10. 01 2010. http://www.latex-project.org/ (Zugriff am 01. 08 2011).

Lippert, Martin, Stefan Roock, und Henning Wolf. *Software entwickeln mit Extreme Programming.* Heidelberg: dpunkt.verlag, 2002.

Madeyski, Lech. *Test-Driven Development.* Berlin Heidelberg: Springer, 2010.

Miller, Ade. *The Purpose of Spiking.* 21. 12 2007. http://www.ademiller.com/blogs/tech/2007/12/the-purpose-of-spiking/ (Zugriff am 25. 08 2011).

Patton, Jeff. *AgileProductDesign.com.* 20. 04 2009. http://www.agileproductdesign.com/blog/2009/kanban_over_simplified.html (Zugriff am 19. 09 2011).

Pichler, Roman. *Scrum, Agiles Projektmanagement erfolgreich einsetzen.* Heidelberg: dpunkt.verlag, 2008.

Poppendieck, Mary, und Tom Poppendieck. *Lean Software Development: An Agile Toolkit.* Upper Saddle River: Addison-Wesley, 2003.

Sillitti, Alberto. *Agile Processes in Software Engineering and Extreme Programming: 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010.* Springer, 2010.

Slany, Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang. *Softwareentwicklung und Wissensmanagement.* 5. 10 2011. https://online.tugraz.at/tug_online/lv.detail?clvnr=144439 (Zugriff am 5. 10 2011).

Royce, Winston W. *Managing the Development of Large Software Systems.* IEEE WESCON, 1970.

8. Appendix I – Pre-Project Survey Results

The following subsections present the survey question groups with their results.

The results are based on a count of 75 fully filled out surveys.

8.1. General Questions

This question group was intended to find out how experienced the participants are with software development.

"Have you already developed software at a company?"

Answer	Count	Percentage
YES	21	28.00%
NO	54	27.00%

The students who answered the previous question with "YES" were asked additional questions then:

"In how many projects did you take part so far?"

Average	7.2
Max	50
Min	0

"Which size were the teams?"

Average	3.9
Max	45
Min	1

"Which software development processes did you use?"

The answers to these questions were quite different. There were ambiguous answers like "All", "Any", "Nothing special" but there were also processes/keywords mentioned:

- Waterfall
- Scrum
- RUP
- Code and fix
- Iterative
- User centered
- Chaotic
- None

"Did you notice some aspects of the project(s) that would have needed improvement?"

- No documentation
- No code documentation
- No coding standards
- No source versioning system
- No processes
- No testing
- Customer feedback
- Communication
- Estimation
- Knowledge management

8.2. Questions about Kanban

This section handles the Kanban artifacts used in the process.

Questions about the Kanban board:

Are the rules concerning the Kanban board clear to you?

Answer	**Count**	**Percentage**

0 (definitely not)	3	4.00%
1	12	16.00%
2	30	40.00%
3 (yes absolutely)	30	40.00%

Do you think that these rules can easily be followed?

Answer	Count	Percentage
0 (definitely not)	1	1.33%
1	15	20.00%
2	44	58.67%
3 (yes absolutely)	15	20.00%

How do you rate the administrative value of the Kanban system?

Answer	Count	Percentage
0 (bad)	3	4.00%
1	26	34.67%
2	32	42.67%
3 (excellent)	14	18.67%

Questions about the Kanban cards:

Is the system regarding the story cards easy to understand?

Answer	Count	Percentage
0 (definitely not)	0	0.00%
1	6	8.00%
2	28	37.33%
3 (yes absolutely)	41	54.67%

Do you think the cards fulfill their function in the employed software process?

Answer	Count	Percentage
0 (definitely not)	3	4.00%
1	14	18.67%
2	38	50.67%
3 (yes absolutely)	20	26.67%

8.3. Extreme Programming and Test Driven Development

Do you think you are able to follow the rules of TDD?

Answer	Count	Percentage
0 (definitely not)	1	1.33%
1	19	25.33%
2	35	46.67%
3 (yes absolutely)	20	26.67%

Do you think TDD can replace the documentation of code?

Answer	Count	Percentage
0 (definitely not)	21	28.00%
1	31	41.33%
2	17	22.67%
3 (yes absolutely)	6	8.00%

Did you practice TDD in an earlier project?

Answer	Count	Percentage
YES	9	12.00%
NO	66	88.00%

Did you practice XP in an earlier project?

Answer	Count	Percentage
YES	3	4.00%
NO	72	96.00%

8.4. Workflow

Is the workflow chart easy to understand?

Answer	Count	Percentage
0 (definitely not)	0	0.00%
1	6	8.00%
2	38	50.67%
3 (yes absolutely)	31	41.33%

Do you think you and your team can stick to the workflow all the time?

Answer	Count	Percentage
0 (definitely not)	5	6.67%
1	21	28.00%
2	41	45.67%
3 (yes absolutely)	8	10.67%

9. Appendix II Post-Project Survey Result

This section gives the detailed results of the survey executed after the project period.

The results are based on 15 answered surveys but not every question was answered by the participants.

9.1. Planning Game

Do you think there can be made a precise estimation of the project velocity after a few iterations?

Answer	Count	Percentage
YES	8	61.53%
NO	5	38.47%

Comments mentioned, that there cannot be spoken of "precise" estimation but nevertheless the estimates are better than with most other techniques.

Does the planning game help to get a better overview of the project?

Answer	Count	Percentage
0 (definitely not)	0	0.00%
1	0	0.00%
2	4	30.77%
3 (yes absolutely)	9	69.23%

Do you think it is important that every team member is involved in the planning game?

Answer	Count	Percentage
YES	12	92.30%

NO	1	7.70%

Comments like "definitely important!" intensified the importance of a planning game for all team members.

9.2. Kanban

How do you rate the administrative value of the Kanban board?

Answer	Count	Percentage
0 (very poor)	0	0.00%
1	1	10.00%
2	3	30.00%
3 (pretty good)	6	60.00%

How important was the Kanban board for helping you in the development progress?

Answer	Count	Percentage
0 (very poor)	0	0.00%
1	1	10.00%
2	6	60.00%
3 (pretty good)	3	30.00%

What did you like about the Kanban-board?

- Project state visualization
- Clear visual feedback of certain states
- Good overview of the whole process
- Easy to read and understand

Is the system regarding to the story cards easy to understand?

Answer	Count	Percentage
0 (definitely not)	0	0.00%
1	1	10.00%
2	5	50.00%
3 (yes absolutely)	4	40.00%

How important were the story cards for helping you in the development progress?

Answer	Count	Percentage
0 (definitely not)	0	0.00%
1	1	10.00%
2	3	30.00%
3 (yes absolutely)	6	60.00%

Do you think the "effort-estimation" value is something to rely on after a few iterations?

Answer	Count	Percentage
0 (definitely not)	1	10.00%
1	1	10.00%
2	3	30.00%
3 (yes absolutely)	5	50.00%

Was there anything on the story card, which needs improvement?

No ideas for improving it.

9.3. Workflow

Do you think the spiking process is useful?

Answer	Count	Percentage

0 (definitely not)	1	10.00%
1	5	50.00%
2	2	20.00%
3 (yes absolutely)	2	20.00%

How important were the paper mockups within your team?

Answer	**Count**	**Percentage**
0 (definitely not)	0	00.00%
1	6	60.00%
2	1	10.00%
3 (yes absolutely)	3	30.00%

Had you any good experiences when spiking?

- It was really necessary because of lack of know-how
- Brings a bit of spare time from pair programming
- Yes, reduces development time. Is nearly as valuable as development.

Did you have story cards where no mockup could be drawn for? What did you do instead?

- Architectural stories or highly dynamic interface details do not lead to a mockup
- Just talk about how something should work
- Write pseudo code
- Explain textually
- There were no story cards where no mockup could be drawn for

What do you think about the "code update – test – commit" workflow?

- Works pretty well
- Was OK
- It is very important to stick to the rules

Were there any problems when sticking to the defined workflow?

Answer	Count	Percentage
YES	6	60.00%
NO	4	40.00%

- Coding just what you need is sometimes not a good idea. Experienced developers should be allowed to "code-ahead" for less work later on.
- Too few code-commits cause difficult branch merging after a time.
- The more team members the more difficult it is to keep them "playing the game".

Any suggestions on how to improve the workflow?

- More know-how on versioning systems
- The more the team members know each other, the more it will improve

9.4. Overall Questions

Did you recognize a gain of quality according to previous projects?

Answer	Count	Percentage
YES	8	88.90%
NO	1	11.10%

Did you recognize a gain of project velocity according to previous projects?

Answer	Count	Percentage
YES	7	77.80%
NO	2	22.20%

Do you think that knowledge is spread well with this process?

Answer	Count	Percentage

YES	9	100.00%
NO	0	0.00%

Would you use this process for other software projects if the team size were adequate?

Answer	**Count**	**Percentage**
YES	9	100.00%
NO	0	0.00%