Hasso Plattner Institut
IT Systems Engineering
Universität Potsdam

Bachelor's Thesis

Agile Software Development in Small Projects

Methods and Techniques used in the Sendinel Project

by

Johan Uhle

Potsdam, June 2010

Supervisors

Prof. Dr. Christoph Meinel

Martin Wolf, M.Sc.

Internet-Technologies and Systems Group

Disclaimer

I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, June 25, 2010

(Johan Uhle)

Kurzfassung

In dieser Bacherlorarbeit reflektieren wir die Anwendung der Agilen Softwareentwicklungsmethoden Scrum und Extreme Programming im Projekt "Sendinel". Wichtige Aspekte sind, wie wir Entscheidungen ohne einen Product Owner gefällt haben, wie wir den Prozess ohne einen Scrum Master verwalteten und wie wir unsere Planung organisierten.

Abstract

In this Bachelor's thesis we reflect on how we used the Agile software development methodologies Scrum and Extreme Programming in the project "Sendinel". Important aspects are how we came to decisions without a Product Owner, how we managed our process without a Scrum Master and how we organised planning.

Contents

1. Introduction

Whenever a team is working together, they organise somehow. There are processes, they follow and rules they obey. Software development is very often done in a team and a lot of ways to organise how to as a software development team have evolved.

This paper will focus on how we used Agile software development methodologies [1] for eight months in a software project with seven team members in a university environment. It will be explained and reflected on how we developed the software "Sendinel" [2] using aspects of Scrum [3] and Extreme Programming (XP) [4].

Software mostly has no physical representation. Not even the user interface is a complete representation of all parts. This makes it hard to track progress and even harder to find mistakes. This can make developing software a very dim process. Therefore a software development process has to assure a level of visibility to the developers and the stakeholders alike.

Software development involves a lot of variables and uncertainty. It is not straight-forward. Change is immanent because requirements evolve throughout development. This makes sufficient up-front planning next to impossible. Nevertheless, mechanisms for planning and controlling progress are needed. Or in other words: Every team has to organise somehow.

There is no general best way to work. Every process highly depends on the situation, the people and the constraints. Agile methods are a tool set a team can use to craft their own process.

We have used them in our project and adapted them to our needs. Some things have worked out great and some still need improvement. This is the normal way when crafting such a process, since you can only learn and improve by actually trying it. This paper will give the readers an insight into our experience and hopefully help them to reflect on their own way of working.

Sendinel is the software developed within this project. It is an Open Source software for clinics in rural areas. In the middle of the project, Sendinel was deployed by us in a clinic in rural South Africa. The actual development was done by the seven students in Potsdam/Germany, over 8000 kilometres away from the users. Additionally to the large distance, communication over telephone or e-mail was complicated and the cultural gap had to be bridged. Thus the software was developed without a strong user involvement.

Apart from these problems, Sendinel was the first time that we worked autonomously on such a large software project. Therefore finding ways of working together was a very important part of the project and one of the key lessons for us as bachelor students. But exactly due to this focus this paper may provide a valuable insight into the development process of Sendinel.

The basic structure of the paper is the following:

- Section 2 explains our project "Sendinel"
- Section 3 briefly explains the software development methodologies XP and Scrum as well as their historical context
- Sections 4 and 5 outline the software development process and reflect on its impact

2. Case Study: Sendinel

2.1. What is Sendinel?

Sendinel[2] is a web application that helps clinics in rural areas to communicate with their patients. It does so by sending SMS and Voice Calls over the standard mobile network without a dedicated internet connection.

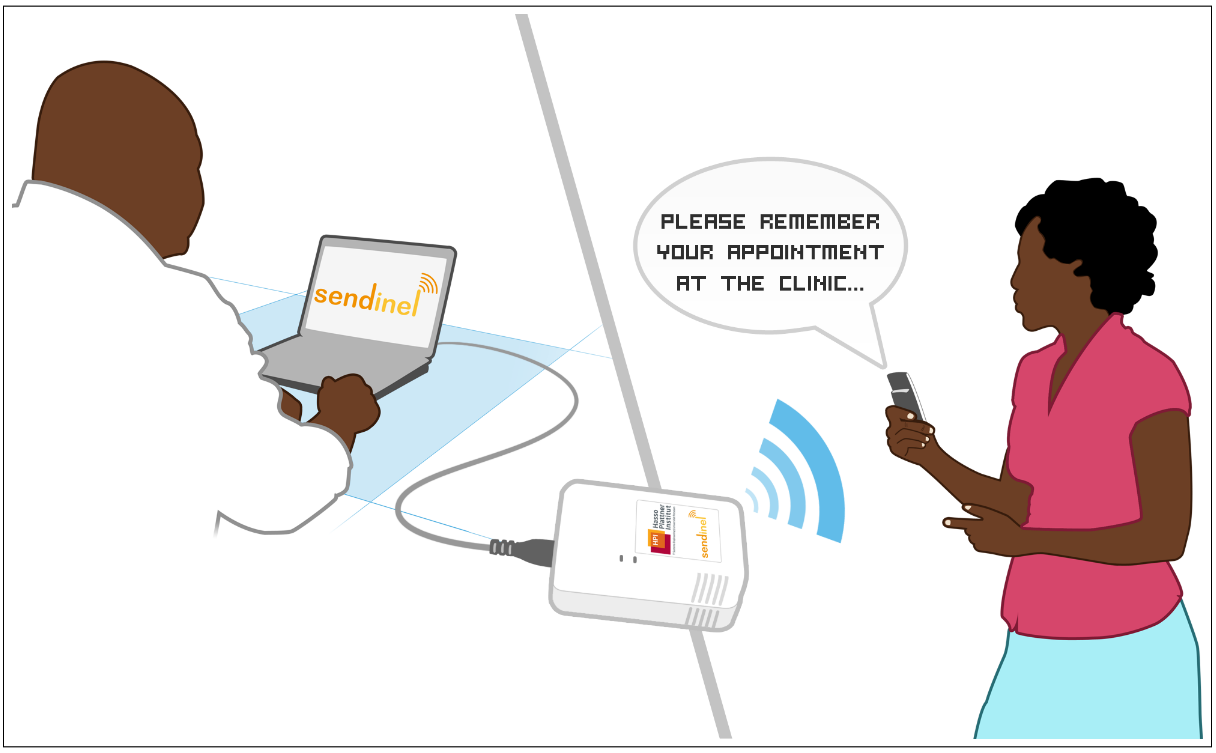

Figure 1: Sendinel: Patients can receive information from the clinic via SMS and Phone Call

The software covers a variety of use cases like appointment reminder, medication arrival notification and bulk messages. It has been developed with a focus on integrating well into the clinic's workflow.

The main application runs on a web server within the clinic and can be accessed by other computers via the local network.

2.2. General Framework of the Project

Sendinel was developed at the Chair for Internet-Technologies and Systems at the Hasso-Plattner-Institut in Potsdam/Germany. We were a group of seven bachelor students who worked independently in cooperation with external partners on the project. It was the final graduation project which marked the end of the bachelor studies.

An office room was provided and we had access to office supply and a technical infrastructure. The project had no self-governed budget but it was possible to apply for cost-coverage to the chair.

The main objective for us as students was to get good marks. The learning experience was how to develop a complex software system within constraints. In the course of the project we learned how to organise ourselves.

Other objectives were to improve the good public view of the HPI and of course to develop something valuable.

2.3. Course of Events

The project lasted eighth months in two semesters. During the first five months we worked half-time since all of us still attended lectures. In the last three months we worked full-time. This plan is also visualised in figure 3 (in the appendix).

The project started with the topic "Healthcare education in Africa". We were given free hand in deciding what to do with it.

From these starting points we searched for problems to be solved during an extensive research and analysis phase. We informed ourselves about the domain and made contact to domain experts. Once promising problems were found we built concepts for them and evaluated them with experts. Furthermore we built prototypes to get a better feeling for the direction we were heading.

At a certain point the concept was concrete enough to implement it. During implementation the concept was constantly adapted to new insights.

Halfway through the implementation the whole team went on a research trip to a clinic in rural South Africa. There the system was deployed under real conditions. We conducted User Research to evaluate the current state the system and possible future extensions [5].

After the research trip we improved Sendinel according to our findings. Afterwards we prepared the code and accompanying material for the end of our bachelor project. Everything is available online under an Open Source License [1]. Furthermore a company in South Africa will probably continue the development.

2.4. Stakeholders

Several parties had an interest in the project and were involved in its success. These stakeholders and their role in the project are described in this section.

The Chair for Internet Technologies and Systems at the Hasso-Plattner-Institut provided the organisational and legal framework for the project. They also provided technical support and gave us feedback.

[1]Apache License 2 http://www.apache.org/licenses/LICENSE-2.0.html

SES Astra was the initiator of the bachelor project. During the project they supported us with contacts and provided a satellite connection for deployment.

The **HPI Research School** at the University of Cape Town is specialised in ICT4D [2] and user interface research. They gave us feedback on our ideas and established contacts in South Africa for us.

In the beginning of the project, we found ourself in the position of not having customers or users at hand. This led to a situation with freedom but also uncertainty. We needed great domain understanding by ourselves. Therefore we researched the field ourselves and established a large number of contacts in science, economy and society. This enabled us to make well-informed decisions.

A contact that grew to significant importance was **SAP Research South Africa**. Their field of interest is how information technologies may improve healthcare in rural areas. With SAP we had access to great domain-specific knowledge and very valuable feedback. Furthermore they offered to organise the research trip to the clinic in South Africa. Thanks to the support of the Chair and SES Astra we were able to accept the offer and conduct the mentioned trip. With this we eventually had access to potential **users** of Sendinel.

[2]ICT4D stands for "Information and Communication Technologies for Development". It refers to the application of information and communication technology to leverage development, especially in poor environments.

3. Software Development Methodologies

The constraints and possibilities of the Sendinel project have been explained in the preceding chapter 2. On their basis we started to think about how we wanted to work. For this the starting points were the software development methodologies Scrum and Extreme Programming (abbr. XP). Both are going to be explained generally in this section. Furthermore the history of how they evolved and what their underlying ideas are, will be introduced briefly.

If you are already familiar with Scrum and XP you might want to skip this section and advance to section 4 "The Sendinel Development Process".

3.1. A Brief History of Software Development Methodologies

Waterfall Model
The waterfall model was the starting point of Software development in the 1960s [6]. Building software is seen as a sequential process consisting of the following steps:

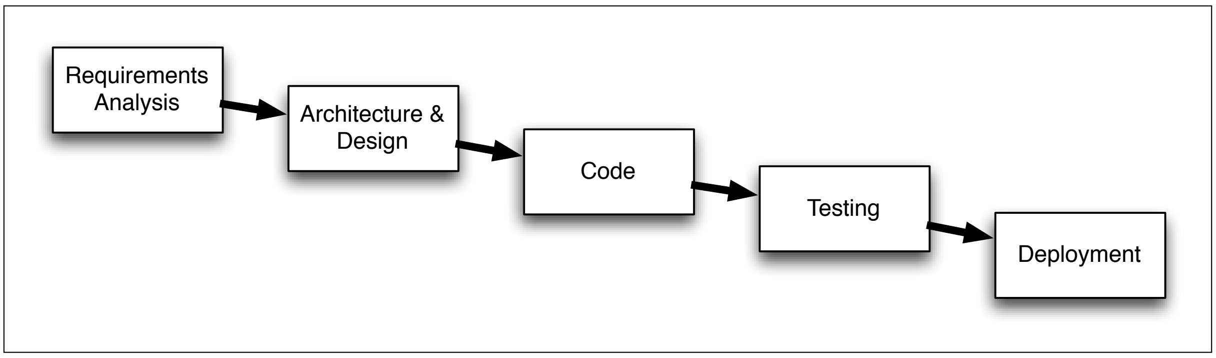

Figure 2: The waterfall model based on [7] Page 2

Experience shows that this does not work, because requirements change over time.

Reasons for that are:

- Software is too complex to be sufficiently described up-front
- Stakeholder do not know what they want. "I'll know it when I see it" ([7] Page 3)
- Technical Problems surface during implementation

A much-used analogy for software development has been the building of structures like a bridge. This manufacturing is strictly sequential. But the analogy is insufficient because of the reasons outlined above. Most importantly, building a structure relies on evaluable physical parameters. Once the calculation is done right, not too much will change during construction. Software development is just too complex to be evaluated beforehand and things will change throughout the process. Therefore sequential development will probably not yield in satisfying results.

Lean Philosophy
The Lean Philosophy derives from the Toyota Production System, which was developed by Taiichi Ohno [8] in the 1960s to improve the car production of Japanese car manufacturer Toyota. It is said to be one of the reasons why Toyota became a market leader in the automobile sector (as stated in the preface of [9]). The main focus of the Toyota Production System lies in the "absolute elimination of waste" ([10] page 3).

This is achieved by two core ideas:

Just In Time Production means only to make and supply what is needed at the moment. Both over- and underproduction are seen as avoidable waste. **Autonomation** means the automatic detection of defects as early as possible. Production may only continue when the defect is resolved. No defective product shall go to the next step in production. No defective machine shall continue to produce.

These core ideas lead to other interesting thoughts:

- Build Quality In - Every step should add value to the product
- Delay decisions to the last feasible point
- Respect the people - Everyone is responsible for the final outcome
- Continuously improvement of the process - Get feedback and include it
- Create and spread knowledge - Avoid monopolies of knowledge
- Deliver fast - The faster the process the better it can adopt to change

Lean Philosophy involves continuous improvement of the process and the product. Even though it started in the productive industry, it was later extended to Lean Product Development [11] as well. From there, the ideas have also been adapted to other areas like Hospitals [12] or Governments [13]. In software development especially Lean Software Development was introduced [14] but the Lean ideas can be found in other Agile methodologies too. For example writing tests is a practise which implements the idea of spotting defects automatically as early as possible.

The Agile Manifesto
In mid-1990s new methodologies like Extreme Programming, Scrum and Adaptive Software Development[15] arose. The persons behind them found that they were having things in common. In February 2001 the Agile Manifesto was signed by 17 representatives of several software development methodologies like XP and Scrum ([16] pages 215 - 223).

The Agile Manifesto consists of four core values:

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

From these values, twelve principles derived [17], which further describe the four values. The Agile Manifesto marks the minimum which all Agile methodologies have in common.

3.2. Extreme Programming (XP)

One of the signers of the Agile Manifesto is Kent Beck. He is also known for introducing Test-Driven Development (citation needed) and is a founding father of Extreme Programming.

XP values humans as most important in the software development process. It puts the customer into the focus of development and accepts adapting to change as a core characteristic of software development.

XP defines a set of values which all participants should agree on. These are Communication, Simplicity, Feedback, Courage and Respect. They are rather abstract and high-level but build the foundation for the principles and practises XP defines. Practises are the concrete techniques executed for working. These practises are the real-world manifestation of values. But still practises and values are too far apart. Principles bridge the gap between values and practises by employing guidelines specifically for the programming domain. Further explanation of the values, principles and practises can be found in [4] Pages 17 - 54.

Some practises of XP are explained next:

The team should sit together preferably in one room. This ensures easy communication and interaction. The general design of the software and its architecture should not happen upfront but incrementally throughout the development. Everything should be revisable and open to change. This encourages an iterative approach. Big parts of work should be cut down into smaller parts. These small parts of work are called stories.

In a planning meeting with the developers and the customer, these stories are defined and estimated. Afterwards the customer can choose, which stories the team should work on in the next iteration. An iteration is a fixed time interval (e.g. one month) after which all declared stories should be done. At the end of every iteration the system should be in a deployable state.

For the actual programming several methods are encouraged:

- During Pair Programming two programmers sit and work together on one part of the code. One is the driver, who is actually writing code. The other is the navigator, who supports the driver.
- In Test-Driven Development the programmer is writing a test, before actually implementing a functionality.
- A Continuous Integration system ensures the correctness of each commit in the code repository [3].

In Sendinel we used all of these three methods, thus they are further described in sections 4 and 5.

[3]This is further described in "Continuous Integration - Practices and Tools used in the Sendinel Project" [18]

3.3. Scrum

Scrum is a framework for agile software development [19]. It consists of roles and practises.

Scrum defines three roles:

The **Product Owner** has a deep domain understanding. Either the customers by themselves are Product Owners or a team member, who is in very close contact with the customers or users, represents the Product Owner. The Product Owner views the software from the customer side and therewith defines the strategic direction of the project.

The **Scrum Master** is responsible for supervising the process. Furthermore the Scrum Master handles administrative and organisational tasks.

The **Team** implements the software and takes part in the planning process.

Scrum defines the following practises:

A **Sprint** lasts from one to four weeks in which the team is working on tasks from the Sprint Backlog. A Sprint starts with a Planning Meeting and ends with a Sprint Review or Retrospective.

The **Product Backlog** is a set of tasks which should be done in the future to finish the project.

The **Sprint Backlog** is a set of tasks which should be done within the current Sprint.

The **Planning Meeting** is attended by all team members, the Product Owner and the Scrum Master. The aim is to define the Sprint Backlog for the next Sprint. Tasks are pulled from the Product Backlog, discussed in the group, estimated via Planning Poker and added to the Sprint Backlog.

During **Planning Poker** a task is estimated. To do this the team members secretly choose their estimation. All estimations are then revealed concurrently. In the following discussion consensus on the final estimation is reached.

In the **Sprint Review** the team presents the results of the Sprint to the Product Owner.

In the **Sprint Retrospective** the team reflects on their way of working.

In the **Daily Scrum** meeting all team members gather to discuss progress and problems that surfaced since the last Daily Scrum meeting.

A lot of methods in Scrum like the Product Owner involvement or the Daily Scrum are very familiar to practises in XP. But even though both methodologies overlap, the general focus of the them is different:
"Scrum focuses on management and organization practices while XP focuses mostly on actual programming practices. That's why they work well together - they address different areas and complement each other." ([20] page 81).

4. The Sendinel Development Process

In this section the ideas used for structuring the development process during the development of Sendinel are described. Included are methods from Extreme Programming and Scrum.

A development process is not defined in the beginning and followed by everyone all the time, but is adapted throughout the project to the current situation and context. Therefore the reader is advised not to see this as a finished methodology but as a work in progress.

An in-depth reflection on the impact of the ideas can be found in the succeeding section 5.

4.1. Analysis Phase

As described earlier in section 2.3 we started the project with an analysis phase. This was necessary because we had no domain-specific knowledge at hand. We had to build understanding for the domain and determine problems and needs. We collaboratively identified tasks and talked about who takes on which tasks. During that phase we followed a very simple process:

- Meet once a week.
- Consolidate the results of last week.
- Define tasks for the next week.
- Talk about who works on which task.
- Optionally make appointments for working together.

This phase gave us time to get to know the group and the new working environment. We also used this time to set-up our non-coding tools (described later in sections 4.6 and 4.7).

After about a month we started to actively work on the first concepts. During this phase we also started to practice peer review. Over time the concept got more concrete and we built prototypes to evaluate the ideas.

4.2. Long-Term Planning

The long-term planning consisted of three Milestones[4] which gave the project a rough structure. Their content was filled iteratively on the way to the Milestones.

In early phase the team had an eye on the long-term in every Sprint meeting. After the research trip and the processing of the results we had a good impression on what to do in the remaining time. Therefore only one final long-term planning session for the last milestone was done then.

The whole project plan is also visible in figure 3 (in the appendix).

[4]A Milestone is a fixed date and to which a certain event should happen. An example is the release of the software to the public at a predefined release date [16] page 119

4.3. Applying Scrum

In January 2010 (for an overview see the project plan in figure 3 in the appendix) the implementation phase started. We decided to "do Scrum". This meant, that we had Sprints lasting two weeks (later shortened to one week). We had no Scrum Master and no Product Owner.

The first Sprint solely consisted of setting-up the coding environments and getting familiar with the used technology.

Before the first Sprint we defined a Product Backlog based on the current concept. This Product Backlog was the basis for early Sprints, where we generated the Sprint Backlog out of items from the Product Backlog. In later Sprints we stopped to update the Product Backlog. Most items either derived from the milestone planning or were already to be found in the bug list or came up automatically, when talking about the teams thoughts on what to do next.

A typical Sprint looked like this:

Sprint planning (Tuesday Afternoon)

- Two people generated a first version of the Sprint Backlog. They looked at the Product Backlog, the bug list, upcoming organisational tasks and the upcoming milestone.
- They also calculated the Sprints velocity [5] for the next Sprint based on last Sprints velocity, probably available man-hours and foreseeable interrupts
- The team came together and discussed tasks in the Sprint Backlog. Some were added or removed
- The team estimated all tasks by doing Planning Poker (see section 3.3)
- The tasks are prioritised by the team in three categories: must-have, should-have, could-have

Acceptance meeting with supervisor (Wednesday Noon)
The acceptance meeting was the groups way of doing the Scrum review meeting. Someone in the team was appointed to be responsible for the acceptance meeting. Prior to the meeting this person checked all the Sprint tasks for their status and instructed last minute bugfixes. In the meeting all done tasks were demoed to the supervisor. Afterwards the new task list was discussed and possibly altered.

Rest of the week
A Stand-Up meeting was done every day (explained later in section 4.4). People continued to work on their pending tasks or pulled tasks from the Sprint boards (figure 4 in appendix) "new" section and moved them to the "pending" section. Each pending task was marked with the name of the persons currently working on it. Once a task was finished it was moved to the "done" section. New tasks were added to the bug list, the Sprint Backlog or the Product Backlog during the Sprint as well, always given it appeared to be reasonable to do so.

[5]Velocity: Amount of work a team can do in a Sprint ([20] page 78)

4.4. Getting Everyone on the Same Page for the Day

We did daily Stand-Up meetings once everybody had arrived in the office. During the meeting everybody explained what they were doing since the last Stand-Up meeting and reflected on problems and opportunities.

Then people decided on which tasks they wanted to work over the day and moved them on the Sprint Backlog from the "new" section to the "pending" section next to their name. The team also tried to set a schedule for the day, clarifying when people can work individually and when the team has to sit together. Furthermore goals were defined on what should be achieved during the day. In the end everybody knew what was going on in the project. This everyday feedback led to a high identification of every team member with the project and a high working morale.

To keep these meetings as short as possible, all team members stood up in a circle and were asked to fully focus on the meeting. Furthermore everyone was asked to be as brief as possible. It was okay to interrupt each other when somebody was taking too long or started to get off-topic.

4.5. Coding and Business Tasks

The goal of the project was to develop software but to achieve this we took on non-coding tasks as well. We had to do a lot of research to gain knowledge about the domain and holding contact to existing sources and building up new contacts was crucial to the project. Furthermore we also had to complete organisational tasks. Especially organising the research journey was a very time-consuming effort. We will further refer to these tasks as business tasks. We decided to include these business tasks in our Sprint planning. The reasons and experiences with that are outlined in section 5.6.

4.6. Physical Tools Used

The team had one office room at the institute. Every team member had a key and was able to enter the room anytime.

The following working environment was provided:

- Own table and office computer for every team member
- Sufficient office supplies (pens, paper, post-its ...)
- Printer, scanner, copier
- Several whiteboards
- One big screen
- Couch (which was removed later)

Private laptops were used as well by some team members.

4.7. Technical Tools Used

The main project communication other than talking in person was handled via **e-mail** on a dedicated mailing list. We also used instant messaging via **Skype** [6] for direct messaging, group chat and video conferences.

We used **Dropbox** [7] for sharing non-code data like documents, pictures and graphics. The most important aspect for us was to have the same data available as fast as possible to every team member. We could have also used a shared drive on a server in the university network for this. Unfortunately this solution did not provide offline access and offered only complicated access from outside the university network. Plain text information like protocols, links and summaries were saved in a Wiki [8].

The source code was managed with the **Git** [9] version control system. The repository was hosted at **GitHub** [10], where repositories are public by default. Since we wanted to publish Sendinel as Open Source software anyway, having the source code publicly available from the beginning was a good idea. Internally we used **Hudson** [11] as a Continuous Integration server.

4.8. Test-Driven Development and Pair Programming

We did Test-Driven Development. The basic flow is like this:

1. Start with a failing test.
2. Write code that makes the test pass.
3. Refactor written code.

The paper "Test-Driven Web Development - A Case Study With Django" [21] further explains and reflects on this approach in the context of "Sendinel".

We also did Pair Programming ([4] page 42). The basic idea is to have two people working together on one workstation. Often at least two computers were used. The driver was writing the actual code on one computer. The navigator used another computer to look up reference.

[6]Skype: http://www.skype.com
[7]Dropbox: http://www.dropbox.com
[8]We used Trac as a Wiki software: http://trac.edgewall.org/
[9]Git: http://git-scm.com/
[10]GitHub: http://www.github.com
[11]Hudson: http://hudson-ci.org/

5. Lessons Learned

In this section we will reflect on our experience with the methods outlined in the preceding section 4. We will also present reasons for some process decisions and discuss alternatives.

5.1. Knowledge Gaps

In the beginning of the project we realised, that a lot of different knowledge levels exist in the team. To even this differences we tried to bring everyone to the same level in our first Sprint where we set-up the coding working environment on every computer and introduce everyone to the used tools. We also held tutorials and defined learning goals.

This preparation resulted in a less-interrupted workflow. It made all the used tools more transparent so that we could concentrate on the actual content.

5.2. Scrum without a Scrum Master

We had no Scrum Master. With at least two points a Scrum Master would have been helpful:

Review process
Our team had no process for doing regular reviews. One big review session was held in the beginning of the project, where we discussed the current situation. The session was very open and rewarding but also very exhaustive. Afterwards we held no dedicated review session again. We spotted several potential reasons: One is, that people want to avoid stress and uncomfortable situations as much as possible. A review is always demanding a lot of energy since it means revealing and dealing with conflicts and actively reflecting on one's own behaviour and actions. Another reason could be that a review takes a substantial amount of time. Since it is not immediately adding value, some might feel that it is a waste of time. Still we believe that pursuing a constant review process would have been good for the team and the project.

A Scrum Master should have pushed to hold reviews. The lack of reviews blocked further improvements in the development process. It would have also pacified tension in the team, which then released itself anyhow but usually in stressful situations. These ad-hoc relief situations were unstructured and did not lead to sustainable process improvement but rather to appeasement.

Moderator for Discussions
Due to the way how we made decisions (as described in section 4.3) we sometimes had very long conversations. Eventually the discussion yielded in some form of agreement but the way there was often paved with unclear assumptions and repeating arguments. A Scrum Master could have streamlines these discussions by moderating them.

A Team Member as Scrum Master?
The question if a Scrum Master should be part of the team or an external party is often discussed (e.g. in [22]). We suggest that for our project a Scrum Master should have been an external party. Everybody in the team was part of the decision process. A Scrum master

has to have a neutral opinion in order to moderate and balance discussion. Thus an internal Scrum Master would not have been able to fully participate in the decision process.

5.3. Scrum without a Product Owner

During the development of Sendinel no customer was present. But in Scrum, XP and the Agile Manifesto customer involvement plays a crucial role. In Scrum the Product Owner is the representative of the customers. This Product Owner is deeply involved in the software development process by making strategic decisions on what to develop next. The basis for this is the deep domain understanding of the Product Owner (this is explained further in section 3.3).

We were not having a Product Owner. Still the responsibilities of the Product Owner had to be realised in the project somehow. As a solution collective decision making was used.

This means that rather through a well-informed ruling by the Product Owner, we tried to come to a decision through team discussion. Members of the team gained domain knowledge beforehand and weighted arguments in the discussion. Final decisions were either made by consensus or a vote with simple majority. This approach required a lot of energy in exhaustive meetings. But on the other hand it also lead to the buy-in of every team member into decisions.

Another alternative is to have a simulated Product Owner in the team. This is someone who is not a domain expert, but has the same responsibilities as the Product Owner, thus is able to make decisions. We tried this approach, but it was abandoned since it provided no advantage over group decisions. This was mainly because a Product Owner needs a bigger domain knowledge than the team to do legitimate decisions that are accepted by everyone. Without this domain knowledge the Product Owner might make too much wrong decisions. In our project bad decisions were corrected by the team. This meant that the team was having better arguments than the Product Owner and thus, had a better domain knowledge. In this situation the simulated Product Owner lost authority and we automatically fell back to collective decision making.

Both approaches show the problems surfacing when no one in the team has deep domain-specific knowledge. A natural level of uncertainty was evident in a lot of decisions we made. We also spent a substantial amount of energy on the process of making decisions.

5.4. Pair Programming

In Pair Programming two programmers worked together on one computer. This lead to collaborative thinking in which the partners found and implemented the best solutions for the problems at hand. It turned out to be very effective because it leads to quality results and shared code ownership. Therefore pairing was highly encouraged and done often.

Pair Programming was very exhausting. Breaks had to be done regularly (e.g. 15 minutes every hour). It is unrealistic to think that one can do eighth hours of Pair Programming on a daily basis. We found that five to six hours per day are a realistic amount. When pairing, driver and navigator should change regularly. Half-hourly changes have proven well for us.

One aspect in Pair Programming is that the collaborative thinking often manifested itself as a lively conversation between the pairing partners. When in one room, people hear the conversations of the others in the background. Alistair Cockburn describes this as "osmotic communication" and a positive effect ([16] page 81). We object because in our experience "osmotic communication" leads to a lot of interruptions which decrease productivity significantly [23]. A solution to this problem would be dedicated rooms for people or pairs who want to work in quite without being interrupted.

Some people might feel that Pair Programming is a waste of time, since two people are doing the same thing one person could do alone as well. This does not match up to our experience. In case that the pairing team members works well together, we feel that due to the decreased error rate and increased peer motivation Pair Programming is more effective than two people working alone.

Of course Pair Programming does not always work. Examples are a substantial skill difference between the pairing partners or are a general dislike between the pairing partners. As a matter of fact not everyone can work together well. To address this people should be able to choose their preferred partner. On the other hand encouraging people to pair with different partners all around increases collective code ownership, spreads knowledge, increases the likeliness of innovative new ideas and builds empathy within the team.

5.5. Communication

We learned that face-to-face communication leads to the best results, even though it is time consuming. This is also reflected in the Agile Manifesto where it reads: "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation." [17].

The other communication channels we used were e-mail and instant messaging. Written communication is error prone because it increases the probability of misunderstandings which mostly comes from the lack of emotion and instant feedback. We had the situation of not knowing exactly what the communication partner wants to tell us several times both in team and with external contacts. Therefore we tried to have personal meetings, video conferences or at least telephone calls as often as possible. This experience is also reflected in literature such as [16] pages 91 - 99.

5.6. Sprint Planning

Finding Tasks
One of the biggest problems we had in Sprint planning was finding appropriate tasks. This mainly derived from the lack of a Product Owner, who normally has a strategic long-term view on the project. We tried to do long-term planning collaboratively in the group but especially in the beginning of the project we just could not plan far ahead since we did not know in which direction we were going.

This led to situations where the Sprint Backlog was not sufficiently defined before starting the task estimation in the group. Thus tasks had to be defined collaboratively during the planning meeting.

We addressed this problem by appointing two people to prepare a draft of the Sprint Backlog beforehand. They defined the tasks and excluded options. While doing so, they thought through the whole project to get a big picture on where the group is at them moment. Of course these two people had a big influence on the direction of the project. To avoid a concentration of power we rotated the people preparing the Sprint Backlog.

During the planning session the prepared tasks were discussed in the group so that everyone still had an influence on the tasks and the direction of the project. Anyhow our approach set track for the Sprint early on and helped us straighten the planning process. In the late project planning also got easier. This was mainly because the milestones and the increasing understanding for the domain gave a clearer picture of the end product to everyone. Additionally discussions were shorter because everyone in the team was driving into similar directions.

Organizational tasks in the Sprint Backlog
As mentioned earlier in section 4.5 we managed both coding and business tasks in the Sprint Backlog.

In the beginning we were not doing that and concentrated solely on coding tasks. But we quickly experienced that the business aspects of the project needed a substantial amount of time. Therefore we needed a method to include the business tasks in our project.

We tried different approaches:

1. Assign persons to business only and exclude them from the planning process
2. Plan a fixed amount of time as buffer for business tasks
3. Plan business tasks equally to coding tasks

The first idea did not work because everyone in the team was to be involved in all aspects of the project. By this everyone was kept engaged and interested in the projects outcome. Secondly the project is in the studies of computer science, therefore every team member should be involved in writing code. On the other hand we did the separation in a weakened form during the first release when we assigned people especially to release-related tasks (mainly public relations work).

The second idea was actually meant only decreasing the teams velocity. As stated before the business tasks varied in needed effort but were predictable to a certain accuracy.

The third approach is only feasible if the business tasks are completely foreseeable and accurately estimable. Adding tasks to the Sprint Backlog within a Sprint is generally discouraged since it invalidates the estimates.

In the end we used the following version, which combines all three approaches:

During a Sprint planning all available business tasks were estimated. Furthermore a buffer for unforeseeable new tasks was included in the Sprint planning. When a new business task arrived during a Sprint we decided if the task would need action within this Sprint or could wait until the next Sprint, where it would be included in the Sprint planning as normal task.

If it was to be treated within this Sprint, the task was written on a special whiteboard. We tried to address every task on this special whiteboard as fast as possible. Having an extra board highlighted the importance of these tasks to everyone. As a result they were finished quickly. Experience also showed that the extra time estimated for unforeseeable tasks in the Sprint plannings matched up with the amount tasks coming in.

Defining Acceptance Criteria for Tasks
One problem immanent throughout the project was defining and controlling acceptance criteria for tasks. This can be boiled down to the question: "Is this task done?". Every time someone moved a task to the "done" section at the Sprint task board this person decided that the task was completed sufficiently. Unfortunately people were sometimes too forgiving with their own work and declared tasks to be done where others still saw aspects to be worked on. On the programming level typical examples were the handling of edge cases or a final clean-up of the new code. Sometimes people also misunderstood the scope of a task and did not implement certain aspects others thought the task would include.

There are several assumptions why this problem occurred:

1. Too small task cards which leave no space for comprehensive acceptance criteria
2. Too much effort to define sufficient criteria
3. Not possible to define sufficient criteria at the moment

All assumptions are addressable. A common approach to fix them would be a person in the team who keeps reinforcing the proper definition of acceptance criteria. This could be a job for a Scrum Master or for a dedicated testing or quality assurance person.

We did the Quality Assurance step during our internal acceptance meetings described in section 4.3 but usually this was already too late since major errors could not be fixed within the running Sprint anymore.

5.7. Keeping Track with Lists

Keeping track of tasks can be challenging. Doing this with lists is the common approach, which basically follows this flow: Write everything that is to be done on lists. Look up your lists when you want to know what to do next. Once completed, remove the tasks from the list. It takes a lot of discipline to keep everything up-to-date and to use the gathered information. Therefore it must be as easy as possible and should integrate well into the workflow. Any unnecessary clutter in the process of maintaining the list or the list itself will result in abandoned outdated lists which have no value anymore.

We followed two key ideas when organising our lists:

Write everything down
Even though the item might be deleted again after five minutes it is still worth being written down.

Make lists visible to all
Alistair Cockburn uses the concept of displays radiating into the room by revealing their

information in a way that passers-by can still read it ([16] page 84). Just by looking around in the room one could see what was currently happening in the project. Team members who searched for their next task to work on would get up, walk around the room and look at the task displays. We also often gathered around a list and discussed its content. During that process we altered the list collaboratively. Tasks which were unpleasant stayed on their particular spot in the list. Every time somebody would look at the list they would notice the task still being on the same old position. Thus tasks would get the needed attention eventually.

We used two tools: Whiteboards and Post-Its.

Whiteboards are great for lists. Everyone can have a look at the list and alter it instantly. If something is done you can cross it out or delete it from the whiteboard. A downside of using whiteboards is the possibility of tasks getting deleted to easily, which may happen by accident while working on the whiteboard. For example a task once got deleted from a whiteboard in the team room when somebody leaned against it during a discussion. Also there is no history. Deleted items can not be recovered. This is especially unfortunate when somebody deletes a task which was not finished yet.

Post-Its were used for keeping track of the product and Sprint Backlog. The post-its of the Sprint Backlog were posted on the windows of the office. Since post-its do not stick well by themselves and have the tendency to just fall off, we taped them to the glass with extra adhesive tape.

One of the downsides of the two approaches is their lack of remote access. The only way to do that is by asking somebody who is currently in the room. In contrast, digital systems offer remote access. In the beginning the team tried to manage tasks with a digital bug tracker [12]. It was abandoned later mainly because managing tasks was very slow. Submitting and retrieving tasks took a lot of effort and had a poor performance when working collaboratively.

5.8. Tests

Tests were one of the foundation of our development process and proved to be absolutely essential. As mentioned earlier in section 4.8 we used Test-Driven Development. This provided us with a big test suite. The tests increased the probability that the software was working as supposed. They also built up confidence in the code. Even though learning how to write tests had a significant learning curve, the benefits surfaced quickly. For example the tests were the foundation of refactoring. During refactoring, one has to assure that no existing functionality breaks. Automated tests are perfect for that. They enabled us to do major extensive refactorings in several hours which would otherwise have taken much longer or which we would not have started without tests because of fear of breaking something.

We only used unit tests but no automated integration or system tests. We also did not test the client-side JavaScript code. The main reason for this was that setting up the new testing environments and learning how to use them was too much of an effort. In the future we would put in the extra effort because this would increase quality essentially [13].

[12]Trac: http://trac.edgewall.org/

[13]For a more refined view on tests in the context of "Sendinel" see [21]

6. Conclusion

Software development is intellectual work. Hence it is based completely on the people doing it. To comfort the people and provide them with the best working environment possible is crucial for developing good software. This counts for big things like the purpose of work over to the planning and work process down to very basic things like the design of the workplace.

Agile software development puts exactly these aspects into focus. They are built on the trust into programmers that they will achieve best results when they find the best way to work on their own. A team can chose from the tool set of Agile methods and adapt them to their own style of working.

This also leads to the discovery that there is no best process. Blindly following the books will not lead to the best results. Instead adapting the ideas of Agile software development to the own situation of the team will do.

For us, using Agile methods has proven to be a good way of developing software. We appreciated the level of freedom they give to each person. The transparency of the process increased confidence in the project. Everyone always knew, where the project stood and what should be done next.

With our software we arrived somewhere completely else than where we started. This is typical for software development and Agile methods respect this aspect highly. They allowed us to change the focus of the software and drive the end-product as far as possible to the needs of the users.

In the following projects we would use Agile methods again. But, in the spirit of the Agile Manifesto, we would possibly not use the exact same process as in the development of "Sendinel" again but constantly adapt and improve it.

7. Acknowledgements

I thank the bachelor project team for the great experiences we had and for working hard but staying sane throughout the project.

I thank everyone who made the trip to South Africa possible.

I thank the Chair for Internet Technologies and Systems for having our project and the HPI for putting software development processes into the study curriculum.

I thank my Mother, Michael and Anton for reviewing the paper.

References

[1] *Agile Manifesto*, 2001.
`http://agilemanifesto.org/`.

[2] Michael Frister, Philipp Giese, Patrick Hennig, Thomas Klingbeil, Daniel Moritz, Johan Uhle, and Lea Voget: *Sendinel*.
`http://www.sendinel.org`.

[3] Ken Schwaber and Mike Beedle: *Agile Software Development with Scrum*.
Prentice Hall, 2002, ISBN 0130676349.

[4] Kent Beck: *Extreme Programming Explained: Embrace Change*.
Addison-Wesley Longman, 2nd edition, 2004, ISBN 0201616416.

[5] Lea Voget: *User Research with spatial distance between developers and users - the Case-Study of Sendinel*.
Bachelor's thesis, Hasso-Plattner-Institut, 2010.

[6] Dr. Winston W. Rovce: *Managing the Development of Large Software Systems*.
In *Proceedings, IEEE WESCON / TRW*, number August, pages 1–9, 1970.

[7] Victor Szalvay: *An Introduction to Agile Software Development*.
Danube Technologies, Inc., 2004.

[8] Taiichi Ohno: *Toyota Production System: Beyond Large-Scale Production (For this theses the german version by Campus Verlag was used)*.
Productivity Press, 1988, ISBN 0915299143.

[9] Taiichi Ohno: *Das Toyota-Produktionssystem, Vorwort von Eberhard Stotko*.
Campus Verlag GmbH, 2005, ISBN 3593378019.

[10] Mary Poppendieck and Tom Poppendieck: *Implementing Lean Software Development: From Concept to Cash*.
Addison-Wesley Professional, 2006, ISBN 0321437381.

[11] James M. Morgan and Jeffrey K. Liker: *The Toyota product development system: integrating people, process, and technology*.
Productivity Press, 2006, ISBN 1563272822.

[12] David I Ben-Tovim, Jane E Bassham, Denise Bolch, Margaret A Martin, Melissa Dougherty, and Michael Szwarcbord: *Lean thinking across a hospital: redesigning care at the Flinders Medical Centre*.
Australian health review, 31(1), February 2007, ISSN 0156-5788.

[13] U.S. Environmental Protection Agency and the Environmental Council of States: *Working Smart for Environmental Protection*.
Environmental Protection, 2008.

[14] Mary Poppendieck: *Lean Software Development*.
International Conference on Software Engineering, pages 165–166, 2007.

[15] James A. Highsmith III: *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*.
Dorset House Publishing Company, Incorporated, 1999, ISBN 0932633404.

[16] Alistair Cockburn: *Agile Software Development*.
Addison-Wesley Professional, 2001, ISBN 0201699699.

[17] *Principles behind the Agile Manifesto*, 2001.
`http://agilemanifesto.org/principles.html`.

[18] Michael Frister: *Continuous Integration - Practices and Tools used in the Sendinel Project*.

Bachelor’s thesis, Hasso-Plattner-Institut, 2010.
[19] Roman Pichler: *Scrum - Agiles Projektmanagement*.
dpunkt.verlag GmbH, 2008, ISBN 3898644782.
[20] Henrik Kniberg: *Scrum and XP from the Trenches*.
C4Media Inc, 2007, ISBN 1430322640.
[21] Philipp Giese: *Test-Driven Web Development — A Case Study With Django*.
Bachelor’s thesis, Hasso-Plattner-Institut, 2010.
[22] *ScrumMaster Is Not Part of the Team | 10 Reasons*.
`http://borisgloger.com/2009/12/03/Ascrummaster-is-not-part-of-the-team-10-re`
[23] Jason Fried and David Heinemeier Hansson: *Rework*.
Crown Business, 2010, ISBN 0307463745.

A. Appendix

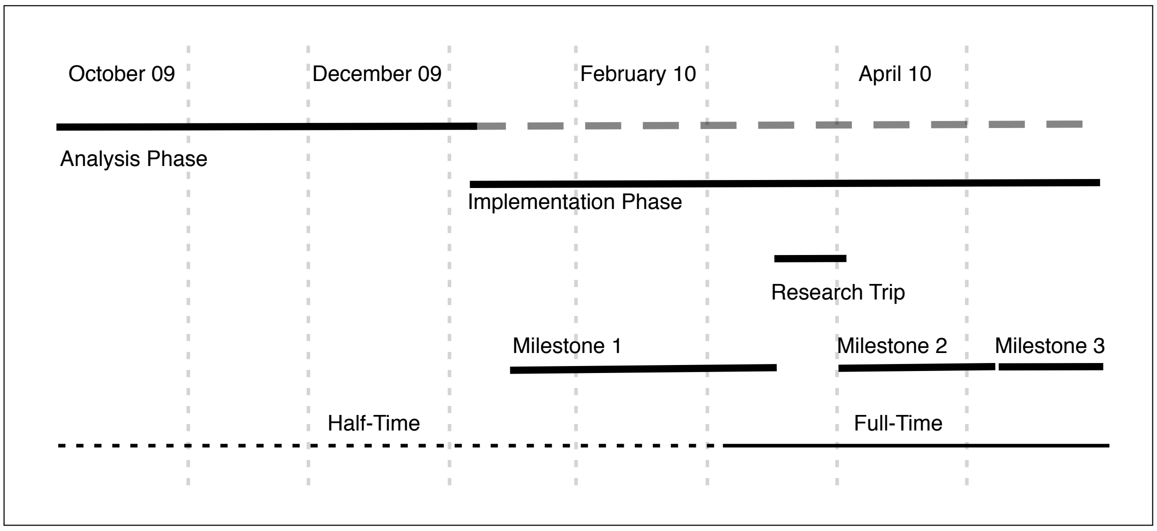

Figure 3: The project plan

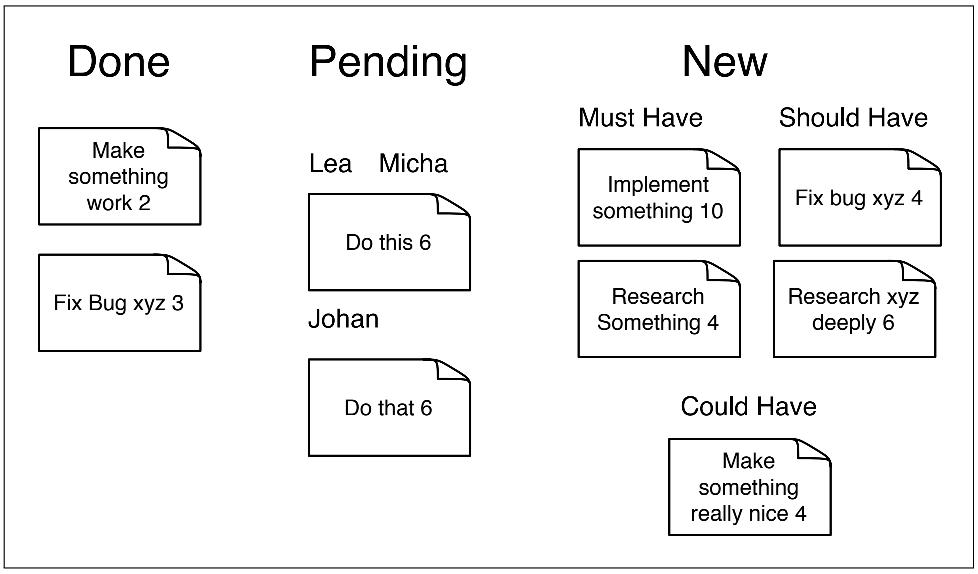

Figure 4: A graphic displaying our Scrum Board with fake task names and estimates. Micha and Lea are currently forming a Pair Programming team. Johan is working alone.