UNIVERSITY OF COPENHAGEN

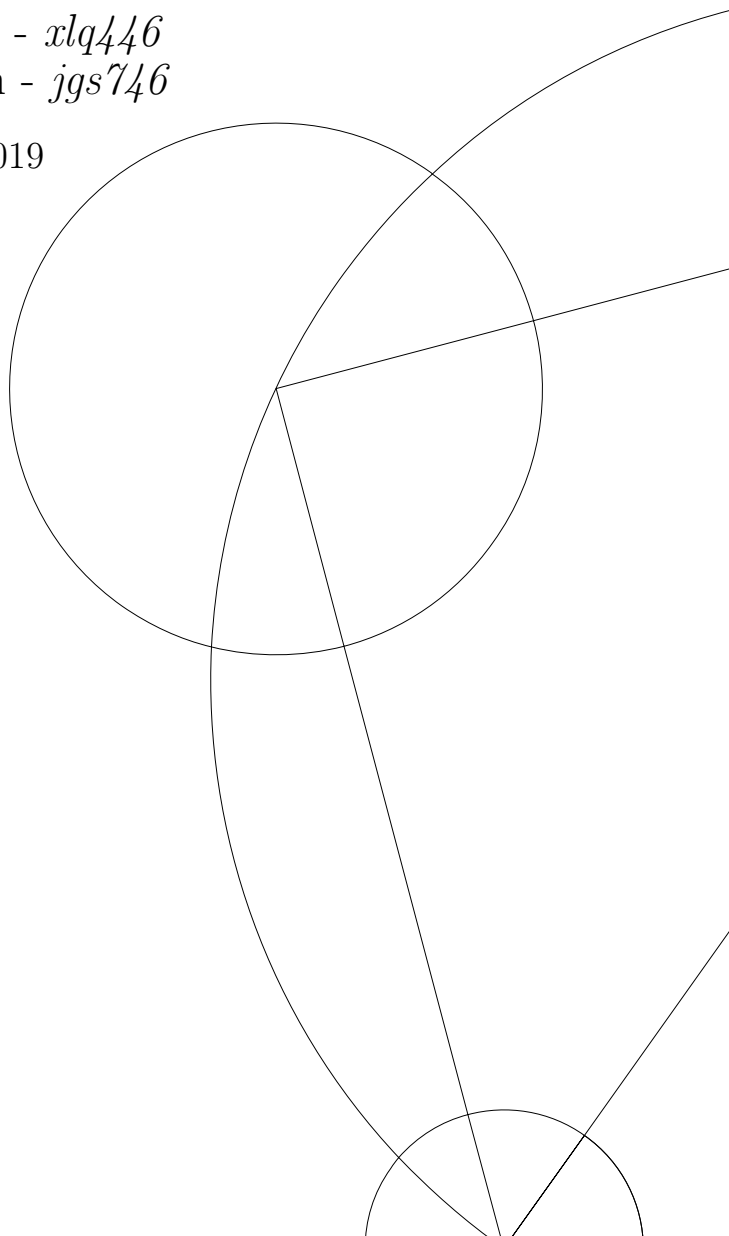
Bachelor's Thesis

Agile Methodologies in small projects

How agile methodologies were used
in the MyTeam.dk project

by
Mads Kronborg - *xlq446*
Niklas Lohmann - *jgs746*

June 23, 2019



1 Abstract

The goal of this work is to achieve a clear understanding of agile software development methodologies and their use in modern software development. To fortify our understanding of the problems associated with agile and how these methods affect software development, we attempted to do a real project. This project was based on the website "Myteam.dk", which we worked on while having multiple interactions with the owner of the site, who was also our client.

We will use this to reflect on how we used the methodologies and whether they helped or hindered us and the project. With a an added aspect being how we came to decisions and how we handled iterations given the size of our team.

While more and more software organizations have lost faith in the old methodologies, the number of agile projects are booming. This paper will try to present the benefits of agile, but will also explore some of the pitfalls, especially the agile project "killer" known as spikes.

Contents

2 Introduction

Software development has for a long time been a risky business. This can be exemplified by many of the well known failed danish projects, for instance the "Tinglysningsprojekt" [11], which went over its estimated time and budget.
Nonetheless the demand of software still continues to grow, and it finally seems like software organizations found a way to to reduce the risk, that software had when using processes like the *"Waterfall Model"* during their development.

With the establishment of the agile approach, software development received a lot of advantages to both the software supplier and the customer. This was mainly as a result of the emphasize on creating workable software, and a focus on reliable communication with the client, which meant that more projects were able to meet the budget and their clients expectations.

Even though agile comes with these benefits, the approach still has its risks. These can come in many forms including the challenge in finding common ground with the customer, and challenges when you have to overcome an issue in a task.
The latter challenge is known as spikes [pages 78-79][3] and is a method used to determine how much work will be required to solve or work around a software issue. During a developing process, spikes help developers to give an estimate of time needed on a problem, and ensures that the team has talked about a possible solution to avoid a dead end.

The goal of this paper is to research and examine how agile development affects the quality of software development, and will attempt to analyze and examine, what makes this method so effective at facing difficulties in software development.
This paper will focus on how we used the agile methodologies [9] for our software project within a small agile team. We will be analyzing and reflecting over our development of the software "MyTeam.dk" using aspects of Extreme Programming (XP) [2] [3] and Scrum [4].

3 Motivation

We, as computer science students, have been interested in this field since we learned of its advantages compared to the old methodologies. Our own experiences of using the agile methods in course-work has led us to raise many questions as to why these methods seem to give projects such a durable development. This project has given us the opportunity to explore the world of agile methodologies to find out the answers to our questions.

Our knowledge of this field at the start of the project was average. We were familiar with the basics of the agile methods, but without knowledge of all the ins and outs. As this is a subject we both were interested in; we wanted to look into why these methodologies made it easier for projects to succeed. Since we had an interest and questions it seemed to be a natural choice of study which we knew would be both challenging and interesting.

The process has been considerably tougher than previous work we have undertaken. Our prior knowledge of using Python and Django has been a great help for us in our process of overcoming a project of this scale of work. Through the development we have become substantially better at everything these languages have to offer, but also at applying this framework to a relational database.

From the theoretical aspect, we have done a lot of research into the agile development and its related areas including the manifesto for agile software development and the principles behind the agile manifesto. We have also done a ton of research into our framework and its different modules, making it possible for us to fulfill our client's expectations.

For this project we would like to work with the following main issues;

- How spikes are handled in agile development? (Agile manifesto, Cohn)
- What are the dynamics for a small agile development team?
- In our project, we implement a new interface in Django, against a relational database. We do this to gain experience with agile development during the project.

We want to educate ourselves on the habits of agile development, gaining experience with the different elements of the agile system development life cycle. Primarily focusing on how "spikes" are handled. Spikes are issues that must be dealt with, and they can come in many shapes, fx a technology issue or a sub project. However it can be difficult to estimate the size of a spike, and a spike could in itself be the considerations on how to handle a difficult task.

We will work on implementing changes to an existing system using agile principles and thereby getting some experience-based knowledge on how spikes are best dealt with in an agile project. During this process we will also be educating ourselves in SQL, the Django framework and web development, which are the tools we will be using during the project.

3.1 Scope

The relational database, which we will be working on, is part of a much bigger project than what we are going to be trying to update/develop. The database is part of a website developed by Hans Jakob Simonesen 2008-2014 and maintained by Anders Lassen. We will only be working on parts of the database and website, since reworking the entire site would be to big of a project.
We will create a list of user stories and determine a feasible subset of these user stories that we will work upon.

4 Software Development Methodologies

Whenever a team begins working on a software project, it's likely that they have decided on a development methodology to follow. The chosen method will give the team a defined set of rules which should be followed and in return the project will gain a framework to help with planning and management during the process.

The main goal of the methodologies is to prioritize the structure for the different phases which should be followed during the development of software. In addition it provides a method to transition between each of the phases. Every methodology includes the basics of the software development cycle, these being:

- Planning
- Designing
- Coding
- Testing
- Maintaining

Basically all methodologies follows these basics, however each model differs in their use of them.

There are multiple software methodologies to choose from, however the difference between them, becomes apparent in how they handle different stages of the project. The majority of current models being used for software development can roughly be separated into two parts; agile development and traditional development.
Agile is the newest approach, known for being iterative, lightweight and associated with only having a few rules and practices, some known examples are Scrum and Extreme Programming.
Meanwhile the older traditional approach is known for being plan-driven and heavyweight, and is associated with a progression of steps; these steps being project requirements, designing the solution, testing and deployment. Examples include the Waterfall Model and the V-model.

In this section we will compare the agile methodologies to the traditional methodologies, while using the method - we used during our own project - as a reference for agile, and using the Waterfall Model as a reference for traditional. We will discuss the advantages and disadvantages for the two methodologies, and try to determine if any theory is better suited for a specific project.

4.1 Traditional approach

The core principal behind the traditional methodologies is that all the needed information required for the project is known before any development. Following the same style as other software methodologies, the traditional approach also follows the fundamental software development practices.

These practices can usually be divided into three steps when following the Traditional approach, where the first step is to plan the entire project ahead by defining the requirements. This is done by using a requirement analysis to specify the design and specifications. The goal is to have the entire system specified before beginning the implementation.
The second step is to define the specifications and the design. This is mainly done with the use of diagrams to represent the design of the system. This gives the developers a base of the system and helps establish a direction for the implementation.
The third step is to implement the software. Developers creates code following the established design from the second step. Usually this step also has some focus on testing, however many of the methods still have them separated into more steps.
The interaction of these phases can be seen on [Figure: 1]. The steps above are the ground layers of the methodology and benefit projects by making it predictive and repeatable.

A mutual element for all the phases in this approach is to have comprehensive documentation. This type of projects usually have more than one team each working on different parts of the project. The goal of the documentation is to be the main source of communication between the teams.
The positives of this approach is that the initial phases leads to the project having a detailed plan and design. This kind of planning is a method that is fitting for a project, in which the requirements are not uncertain.

Now, the first issue software projects often face - when following the traditional approach - is having a hard time meeting the customers expectations. The reason for this is that it's very challenging to capture all the expectations and requirements from a customer in the beginning of a project. This is also magnified by the fact that the customer often doesn't have any knowledge of software, and the development team have to use their past experience and expertise to examine what the customer actually wants.
Another reason which amplifies the problems with the traditional software approach, is that the customer wont see the system before reaching the end of the project. This also means they wont be able to describe the changes that they require for the project to fulfill their expectations until it might be too late.
However many of traditional approaches are attempting to fix these problems, which is visualized on [Figure: 1], where adding the green and blue together is known as the modified version of the Waterfall Method. Compared to the normal waterfall this approach has a *validation* and *verification* phase introduced between each of the phases. This helps the developers in catching any deviations which makes it possible get it changed immediately. In addition, this approach is able to go between the phases if they should find something at later stages.

While this modified version undeniably is better than the original version it's still very hard for a customer to understand the design of the system and often the customer is not provided with a realistic and representative presentation of the end result.
Combining these factors with the fact, that it's hard for the customer to request project changes, as a result of the detailed upfront specification. In conclusion, the traditional approach is best used in an environment where the customer has a clear vision of its requirements and expectations. In the next section we will go in depth with one of the methods following the traditional approach.

4.1.1 Waterfall

Waterfall was one of the most popular traditional methods, but the method was not flawless. On the contrary it had it's share of problems, among others, having no iterations. This is partly why the original waterfall approach is no longer among the most popular, however the model has changed with the times trying to implement some sort of iteration; This is known as the modified waterfall approach.
In the original waterfall approach the development is represented by a series of steps which are to be completed during the project. The order of the steps is crucial as the next phase cannot begin before the prior one has been completed. In the waterfall approach testing has been abandon in the implementation phase, instead it happens after the implementation phase and therefore it doesn't look at individual parts of the system; instead the waterfall approach looks at the full project. This often leads to massive test cases which often means that some crucial test are missed.
During each step of development the result is compared to the requirements created in the beginning of the project. Each of the phases can only be regarded as complete when all the set requirements are finished. On the figure below, the process of waterfall can be seen; where blue marks the traditional while blue and green marks the modified. [Figure: 1].

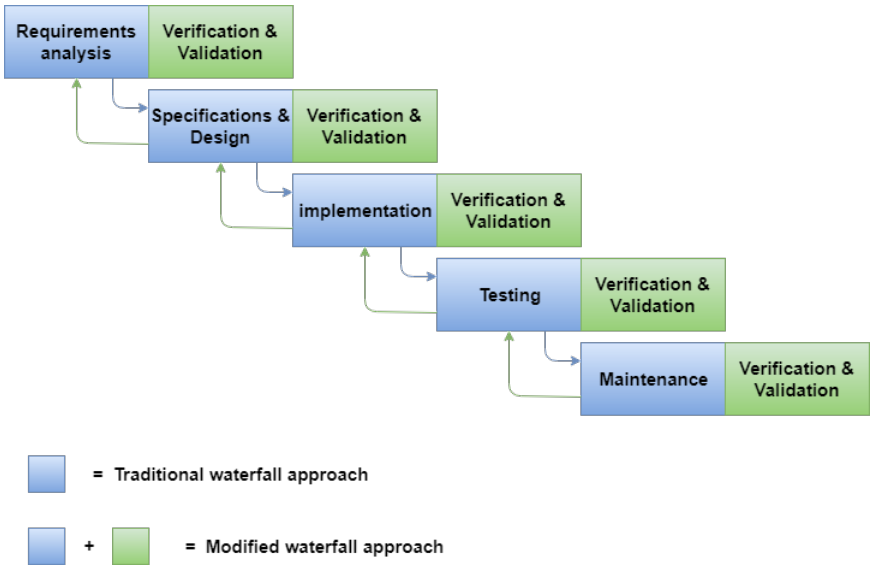


Figure 1: Traditional approach: The Waterfall Model

Our figure shows the step by step development process of waterfall approach. It can be mirrored to the general software development cycle as we explained in the section above. Where the first phase creates the user requirements for the project, afterwards you design the project. Next is the implementation where the developers code the project. The coding leads to testing of the project; making sure that the software is efficient and usable, which leads to maintenance which is where the software is deployed and kept updated.

The waterfall approach is one of the oldest models for creating software, though this also means that many of the problems that comes with the traditional approach is represented in this approach. We explained a lot of the problems in section 4.1,however the biggest flaw is that the project can't be changed after the requirements are completed. With long software project, the requirements will change constantly, which leads to the conclusion that the waterfall approach isn't fit for bigger project where the requirements can't be fully decided in the beginning.

4.2 Agile development:

Agile development is a common term for a number of software development methods, where the emphasis is on following the four values of the *Manifesto for Agile Software Development.* [5](pages 215-223), [9].

In our appendix is the full description of the manifests, together with a explanation. [The 4 Manifesto], however the four core values can be described as having the following focus:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

It was from these values that the 12 principles derived. We have described these more in our appendix at [12 principles]. We will refer to the appendix whenever our task follows or completes one of the principles.
To clarify, these principles are used to further describe the four core values. In our project case we weren't able to benefit from all the principles, as some of them struggles to work in smaller teams, the ones we made use of and thus seen as the most essential for working on an agile project in a small team, were:

- 1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- 2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- 3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

- 6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- 7. Working software is the primary measure of progress.
- 10. Simplicity–the art of maximizing the amount of work not done–is essential.
- 12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

To conclude the main differences between traditional and agile is that traditional only has a working product at the end of the project, meanwhile agile has a focus on incremental deliveries, this method allows the project not to be planned completely ahead and therefore details can be changed as the development continues.
This helps minimizing risks as the members of the project can postpone tasks until more knowledge is available, at the end of each iteration the customer should be presented with a working example of the software. The iterative method also welcomes changes from the customer, which helps agile reaching the goal of fulfilling the customers expectations.

We gave an example of the traditional methodologies introducing some of the elements from agile, in the form of the modified waterfall approach, this is a pretty good indication of the fact that agile is doing something right, since the older methods are starting to adapt processes.

4.2.1 Extreme Programming (XP)

For our project we made use of elements from Extreme Programming which is also known for most contributing "pair programming" and "test-driven development" to the agile practices. It's created by Kent Beck, whom also is a signer of the Agile Manifesto.

One of the most important qualities of XP is to have the customer as a focus in the development, which means that the developers should accept requests from the customer; and adapting to these changes, is the core characteristic of agile software development.

Another important aspect of XP is its predefined values, which all members should agree on. These values are heavily associated with the agile principles and are "Simplicity", "Communication", "Courage", "Respect" and "Feedback".

It can be hard to understand these values since they are rather abstract, however they are the groundwork of the principles and practices of XP. The principles and practices are the techniques which the developer's work on to complete the abstract values. Further explanation of the values, principles and practises can be found in [2][Pages 17-54].

An example of a XP practice is, during the planning meeting the customer should work together with the developers to define user stories. These user stories are then picked in collaboration with the two. It ends up with the team having some user stories to work on in

the fixed time sprint. After each sprint all user stories should have been worked on and seen some progress as well. Also the system should be in a deployable state. This gives us a very circular planning and feedback loop as [figure 2]. Later in the project we will refer back to these practices and values, when explaining the development of the project.

Figure 2: Planning and feedback loops in extreme programming. [12]

Lets go over the figure starting with **Release plan**. This is where decisions are made for the approaching release. This stage usually involves three phases, these being:

1. Exploration Phase, where the team will work together with the customer to create user stories that describe the goal of the system.

2. Commitment Phase, has a focus on planning and improving the team for the next schedule.

3. Steering Phase, where the team tries to accommodate for the changes which the customer request.

Next is the **Iteration planning**. It consists of the same phases as the release plan, however with variation in its goal:

1. Exploration Phase, during this phase some of the customers requirements are chosen to be part of the current iteration. This phase also marks the start of the iteration.

2. Commitment Phase, planning how to complete the current stories during the sprint.

3. Steering Phase, this marks the end of the iteration and the result is compared to the the starting user stories.

Next is **acceptance test**. These are created together with the customer. They are used to validate an expected result from the system.

Then we have **stand up meeting**. This is a daily meeting which assures that all members of the team are synchronized.

Which leads us to **pair negotiation**. This is where the pairs for the upcoming pair programming must negotiate a common share plan of action, making sure that they have shared goals and plans.

Then comes **unit test**. These are created by programmers, making sure that any changes to the code doesn't affect other areas.

In the end we have **pair programming**. This is where the two programmers - decided in the negation phase - starts working together on one computer, which gives **code** that are ready to be used in the system.

4.2.2 Scrum

We also made use of elements from Scrum, which were created by Jeff Sutherland [4]. It's the most representative of the Agile Methodologies. The main reason for this is because of its simplicity. This methodology consists of only three roles:

1. **The Team**, they have the task of implementing software.
2. **The Scrum Master**, have the task of supervision the different processes making sure everything is handled in time.
3. **The Product Owner**, can be the customer or in some cases it can a be a team member that represents the customer. They have the task of viewing the software to see if it lives up their representations. They decide whether the project is going in the right direction or if they have elements which should be changed.

During the development of a Scrum project, you will end up following the practises as seen on figure [3].

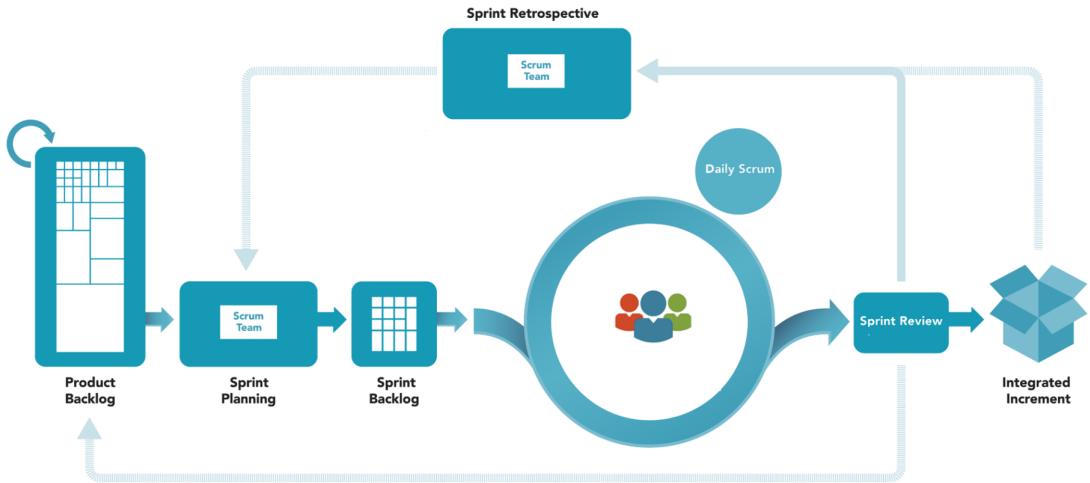


Figure 3: The Scrum process. [13]

Starting with the **product backlog**, the backlog is all the user stories compiled into a compendium, which the team has developed with the help of the customer. These are the stories which the team seeks to complete by the end of the project, for the project to be completed.

Following the arrow to the next task **sprint planning**. Here all roles are represented, and their task is to define which of the user stories from the product backlog that should be pulled into the sprint backlog for the next sprint. Whenever a user story is considered it's discussed between the members.

Then we come to the **sprint backlog**. As explained in sprint planning this is the place for all the user stories chosen for the current sprint. It contains all the tasks that should be done by the end of the sprint.

Next is the **sprint**. In our case a sprint were 3 weeks, however the time can adjust from team to team. This is the phase where the team works on the tasks from the sprint backlog.

During the sprint exists **the daily Scrum**, where the team meets and discusses their progress on their current user stories. Additionally, this is often the time when problems are being discussed and solved.

After the sprint comes the **sprint review**. This is where the teams meets with the customer to present their sprint results, and see if it meets the customers expectations.

Last task of a sprint is the **sprint retrospective**, where the team conclude the sprint, and where they figure out if they have learned anything for the next one.

To conclude, our chosen frameworks have a lot of similarities in their practises. This isn't necessarily a surprise since both of them try to follow the principles of agile. They have different objectives, with Extreme programming having a focus on programming practices, while Scrum has a focus on management and organization practices. This is also why the two frameworks work so well together and why so many projects chose to pair them. [page 81][7]

5 Thesis project: MyTeam.dk

In this section we will describe the project, which problem our client described and how we used agile methods to develop it. We will focus on which agile methods we used and how they helped us complete the agile principles and whether they were helpful or not.

5.1 What is MyTeam.dk

MyTeam.dk is a website that helps teams with planning. It does this by making it easier for the teams to communicate with each other so they easier can coordinate for events.

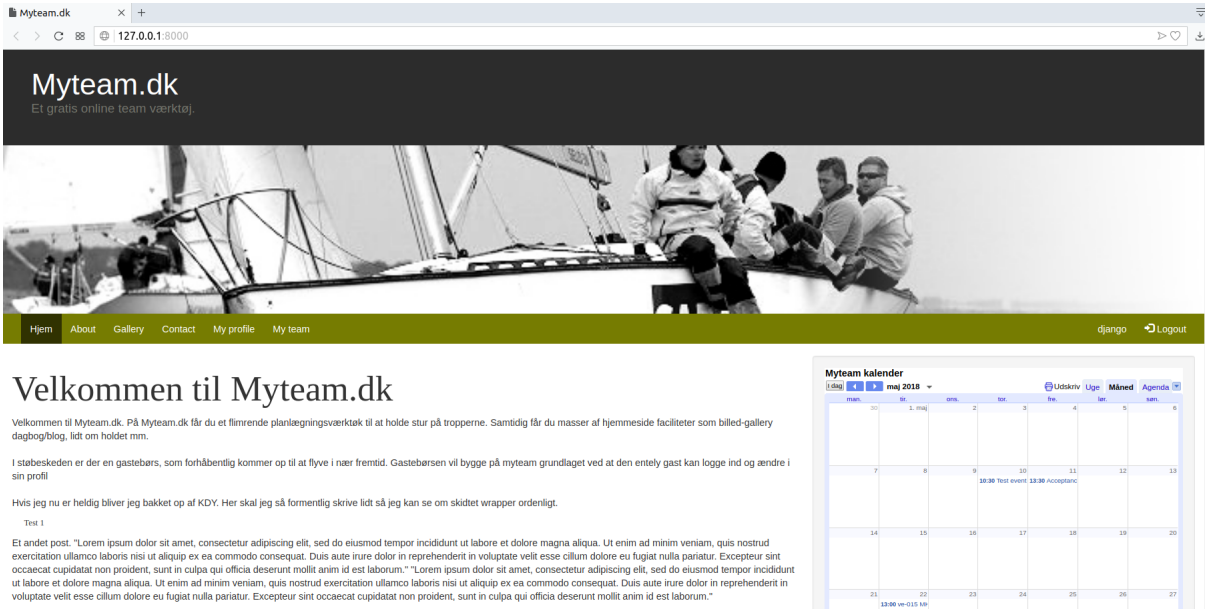


Figure 4: Front page of the new website

The software covers a variety of user cases like event reminder, payments and the possibility to blog. It has been developed with a focus on reassembling the old website which no longer is supported.

5.1.1 Sprints

When we decided the length of each sprint we agreed that three weeks would be the optimal solution, since it gave us one week for information gathering and two weeks for development. Our meetings with our client didn't have the same routine during the project, but it was usually in the end of the sprint, however we also had some at the halfway point.
This meant that we could summarize the results of the sprint with our client, and instantly start the planning of the next one.
With the halfway meetings we were able to summarize the status of the current sprint, which usually meant discussing the current situation. Here we discussed if a task were to be changed or even removed.

5.1.2 Backlog planning

For an agile project the most important part of planning is the product backlog, which is an ordered list of everything that's known to be needed in the product. In agile these requirements normally comes in the design of user stories. [pages 5][1].

User stories are a short deception of a feature told from the perspective of the person who desires the feature, they generally come in the template form:

$$\text{As a} < \text{user} >, \text{I want} < \text{feature} > \text{so that} < \text{reason} >$$

We constructed our user stories together with the product owner to find appropriate tasks. Since the owner normally has a strategic long-term view on the project, he might have some requests that we wouldn't be able to think off. This way of planning instinctively follows the third manifest since it works on customer collaboration. The planning of user stories were successful and we ended up with a long list of requirements that we could work with (See appendix 10.5).

Next was to plan how the user stories were going to fit within our 4 sprints. We did this by having a meeting before each sprint with the product owner, where we defined which of the user stories from the product backlog we would work on in the current sprint.

We then took the chosen user stories and worked with them as shown on the figure 8. This process is how we managed the user stories for the sprint; evaluating whether the story was to large or a spike and had to be split into smaller ones, or if the user stories were to small and could be combined with others. Finally we also had to discuss whether a user story was out of scope and shouldn't be implemented in the current sprint at all.

This figure leads itself naturally to handle spikes, by using the very agile idea of going back and reevaluating a task. For instance most of our spikes encountered during development and testing phases, we then took these spikes and reevaluated among ourselves for a possible solution.
This could then lead to the spikes being split into new user stories - which we then could work on in a new sprint - or we could figure out an alternative plan which fulfilled the user story and then try that solution instead. If we weren't able to find a solution that completed the user story or if our new plan was to different from the original we would wait with the task until we had a meeting with our client. Then we could explain the situation to our client and ask for his opinion before we made a decision.

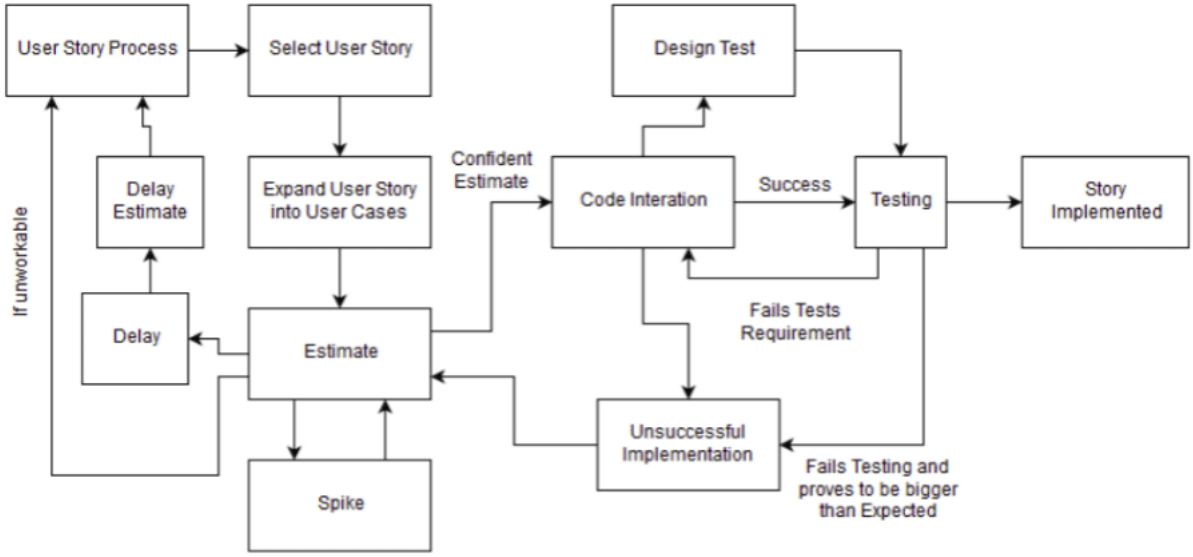


Figure 5: Process to select and manage user stories. [8, Project Luna]

5.1.3 Scrum used in the project

With the creation of our backlog, we were able to move on, to one of the most influential mechanics of the Scrum methodology; the Scrum task board. For our scrum board we followed the traditional model of an agile board as described in [4, page 272], however we removed the review phase on the board, since our review of the software usually happened in testing and in meetings with the client. In our case the board was a whiteboard located in the room where we met with our customer, and it followed the format as seen on the figure below.

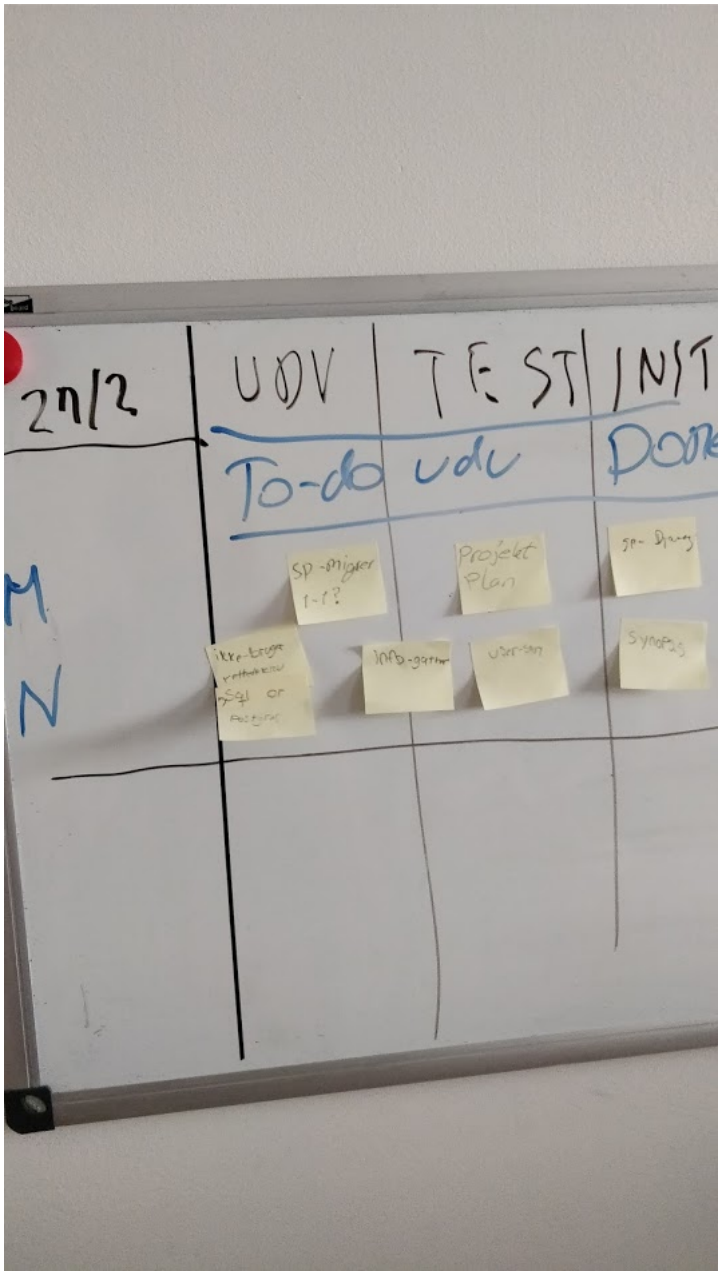


Figure 6: Scrum board during the first sprint

This method gave each team member a box with their current task. We then worked on these tasks going through the columns from left to right. Each of the columns are described below:

1. **To-do:** The user stories that had been approved for the sprint and had to receive work.
2. **Development:** The user stories that are being worked on and receiving testing.
3. **Done:** The user stories that have been completed during the sprint.

At the start of a sprint all the chosen tasks would be contained inside the *"to-do"* column. They would then be moved depending on their progress. If tasks weren't completed during the sprint, they would be reevaluated, which usually means that they either are split into smaller tasks or removed.
This helped us to minimize the risk of tasks being beyond our capabilities by either simplifying the task, thereby following the tenth principal of agile [10. principal][10.4] or removing them since we evaluated them to be out of scope for the project or we found an alternative that our client were happy with.

Seeing back on the project our board was quite successful. It made sure that the client saw our progress with the software, which helped us achieving the first principal. [1. principal][10.4].
In addition it also helped us achieving the sixth principal [6. principal]10.4. Since each meeting started at the board with us describing and showing the progress we had made, we automatically had face to face conversation.

The board made sure that we were transparent with each other and the client. This meant that whenever we encountered a spike, we were quick to communicate it and find a solution. Transparency is one of the most important elements of Scrum as described by Jeff Sutherland.

One element of Scrum that's often a prelude to achieving autonomy, mastery and purpose is transparency. The idea is that there should be no secret cabal, no hidden agendas, nothing behind the curtain. Far too often in a company it isn't really clear what everyone is working on, or how each person's daily activity advances the goals of the company.[4, page 268]

One of our alterations from a normal scrum board is that we didn't have one task assigned to a developer at a time. Instead we chose between all the tasks attached to the sprint and split them between our two man team.
This method gave us a high risk of overloading a developer in the case of unforeseen spikes, while we were good at communicating with each other, we still acknowledge that it would be a safer method to have only one task assigned to a member at a time.
It would however be hard to keep a whiteboard up to date, when we didn't see each other daily, however a possible solution could be to use one of the many online alternatives. This would have made it possible for us to monitor the task flow. In addition to this, in an online solution, we wouldn't be limited by the size of our note and could therefore have additional information about each task.

6 Pair programming

For this project we made use of pair programming, which is a method where two or more programmers work together on a single computer. Which presents collaborative thinking between the two programmers, and often leads to a working solution for the current task. During our project, pair programming was especially used during backend work. At the end of the project we started working individually but this we will come back to later.

Working with pair programming is a very draining task. To combat this we regularly took 10 min breaks, and we also changed driver and observer regularly, to keep ourselves engaged on the coding. The reason for our fatigue, could be caused from us taking to long segments at a time. We did try different time lengths and definitely felt that shorter segments around (2-4 hours) worked better than longer ones (6 hours+).

An aspect of pair programming is that collaborative thinking often evolves into a conversation between the partners, which in turn raises the engagement of everyone in the room. This is described by Alistar Cockburn as "Osmotic Communication":

"The strategy of Osmotic Communication suggests that people who need to communicate a lot should sit so they overhear each other in their background hearing. They learn who knows what and which issues are current. They can respond very quickly to the information. Flow around them. " [page 19][5].

A lot of people see this aspect as a negative since the conversations can cause interruptions and decrease productivity. Alistar Cockburns solution to this were to create dedicated room for different tasks [page 23][5]. We can definitely see the benefits in this approach. We believe such a solution could have helped us avoid fatigue when working with pair programming, since you have the option to relax when working on a long pair programming segment.

We acknowledge that this strategy definitely works better when working on pair programming within bigger teams, and we mainly used the "Osmotic Communication" approach when working with our client. This meant that one of us could continue to work on other aspects while still getting information about the task at hand.

During our development we especially made use of pair programming for our backend focused sprints, our experience was that pair programming decreased errors and increased motivation between the partners early in the process. Though as stated before, when the process came to the later stages it came with the downside of fatigue.

Later on, in the project we strayed away from pair programming, as we felt that we completed more user stories when working individually. This was however as a result of the user stories being less complex, and we still worked with pair programming when working on user stories that was determined as spikes.

To conclude our experience, we perceive pair programming as a helpful tool when working with agile projects. Especially when developing a spike, a team can have an easier time solving it than a single individual.
We realize that pair programming have a position in smaller agile teams, however we weren't able to benefit from it as much as we would have liked. The reason is that we were lacking discipline, and this combined with the lack of experience meant that we often ended up being frustrated.

7 Django:

Django is modeled around a Model-View-Controller (MVC) framework. MVC is a design pattern which aims to separate a web application into three interconnecting parts: [10]

1. The **model**, which provides the interface with the database containing the application data;

2. The **view**, which decides what information to present to the user and collects information from the user.

3. The **controller**, which manages the business logic for the application and acts as an information broker between the model and the view.

However Django uses some sligthly different definitions, which the creators felt were more fitting for the framework. They describe it as an "MVT" pattern, MVT being an acronym for "Model-View-Template". [10]

1. The **model**, is functionally the same. Django's Object-Relational Mapping (ORM—more on the ORM later) provides the interface to the application database.

2. The **view** manages the bulk of the applications data processing, application logic and messaging.

3. The **template** provides display logic and is the interface between the user and your Django application.

7.1 The Django ORM

The Django ORM is an implementation of the 'Object-Relational Mapping' concept. The purpose of ORMs is to provide a bridge of sorts between relational database tables and Python objects. We briefly explained some of the benefits of this concept, in the section about the Django models, but in this section we will dive a little deeper into how this 'bridge' works.
The translation between the framework and the database is done with the Django ORM, but the actual bridge lies hidden within, and for our project, it's 'psycopg2', which is a PostgreSQL database adapter for Python. Django is however not limited to just one option, and thus you have to specify which one you want to use.

We do this in our `settings.py` and for our project we describe our database configuration to Django like so;

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'djangodb',
        'USER': 'django',
        'PASSWORD': 'django',
        'HOST': 'localhost',
    }
}
```

7.2 Django Models

"A Django model is a description of the data in your database, represented as Python code. It's your data layout – the equivalent of your SQL `CREATE TABLE` statements – except it's in Python instead of SQL, and it includes more than just database column definitions." [10] Django uses the models to execute SQL behind the scenes and return pythonic data structures, which represents the rows in your database tables.
If you are a developer with experience in SQL, you might ask why you wouldn't just use regular SQL?
There are multiple reasons why Django is designed like it is, but some are described well by Nigel George in his book, like this paragraph on introspection;

> "Introspection requires overhead and is imperfect. In order to provide convenient data-access APIs, Django needs to know the database layout somehow, and there are two ways of accomplishing this. The first way would be to explicitly describe the data in Python, and the second way would be to introspect the database at runtime to determine the data models. As introspection adds an unacceptable level of overhead and can be inaccurate, Django's developers decided the first option was the best." Among other items, he also mentions that having everything written in Python limits the amount of times you have to make your brain do a context switch. And in limiting that context switch, it helps the productivity."[10]

The drawback of having everything done in Python, is that you manually have to make migrations from Django to your Database. So if you forget to do so, you might end up having your Python code out of sync with what is in your database.

7.3 Django Views

A django webpage can be split into 2 parts, the django-url and the django-view. Any webpage needs an url, and just because the webpage is written in Django, doesn't make it an exception. All the different urls of a project is contained within the `urls.py` file, and when a new project is started, the urls file looks like this;

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Looking at the code we see 2 parts, different python imports, that import Django extensions, and the "urlpatterns". The imports aren't very important, but the urlpatterns is. Urlpatterns is just a list, but it's a list of all your `url()` instances, and these instances is the mapping between all your different pages. But an url is just that, an url, a link to a webpage. The actual code that handles this url and how it's presented is done in the corresponding view.
The most basic view possible, is a function that takes a request, and simply returns a rendering of a html page. These are used for pages which do not have dynamic data or any functionality other than presenting the relevant html page.

```
def about(request):
    return render(request, 'about.html')
```

Later in the report we will further describe how more advanced views can look and act like, by explaining - through example - using our own views from the project.

7.4 Django Templates

The Django template, might appear to be just like regular html, and it can be, but it has potential for much more. This is because of the "Django template language"[14] A Django template can be of any text-based file format (html, csv, xml), and it's designed to feel familiar and comfortable, to those who are used to work with html. What makes a Django html template, different from a regular html file, is that the template contains variables that will get parsed/evaluated by Django and then get replaced by values. On top of this, the template also contains tags, which control the logic of the template.
A variable is written in the template as `{{ variable }}` and this is evaluated in the template engine, which replaces the variable with the result.
Taking an example from the documentation, `{{ section.title }}` will be parsed into the template engine, and it will look at the 'section' model, to find the 'title' attribute, and

return the contents of this field.
Next is the `tags`, these have a similar syntax to the variables, but have added %'s to differentiate them. Tags is what really makes the template language, feel like an actual language. Tags allows for the use of `for` loops, `if/else` statements and having these functions in combination with the variables it becomes a very powerful tool. They use the following example in the documentation;

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

this block of code does the following:
The ul tag at the start tells us, that we are making an unordered html list,
and for every athlete(athlete being an instance of their athlete Model), in the athlete_list, make a list item with that athletes name.

Like with views, we will dive further into templates by looking at our own examples from the project, later in the report.

8 Sprints

8.1 First sprint

8.1.1 Vision

When planning our first sprint we looked upon our user stories and decided which ones would be most relevant for the first sprint.
We decided that our first focus would be on the non-user-defined requirement, ns01 "migrate mysql DDL(data definition language) to postgres/Django" in addition to this we would also look at the user stories 6 through 10. The stories focusing on reimbursement of expenses.

8.1.2 Implementation

Setting up our development environment was the first task at hand. IDE's was not important since neither Python3, Django or Github relies on any specific editor. Therefore after setting up and installing all our different packages and environments, the biggest task for our first sprint was migrating the MySQL database to Django. This migration was a multiple part job, and had two possible solutions, we could either migrate it to Django and then use Django to generate the PostgreSQL, or we could reverse the process, and convert the MySQL SQL-script to POSTGRES and then generate Django `Models` from the SQL tables. We decided to go with the latter.

Migration
The migration from MySQL to Postgres, was not as straightforward as one could hope. There were many conversion issues both in syntax and data types. To name a few we have examples such as;

MySQL	Postgres
Int(), int(8), int(4)	integer
tinyint(4)	smallint
blob, longblob	BYTEA

If we compare the two types of SQL, a statement in each language could look like this;

```
1 `persID` int(11) NOT NULL,        #MySQL
2 persID integer NOT NULL,          #Postgres
```

Django → Postgres
Our first approach was to rewrite the Mysql into Django models, and then use the Django ORM to generate the Postgres tables for us. This plan had the flaw of not including many of the relational database functions, such as foreign keys, constraints etc. These functions could be implemented in Django, but would require a higher understanding of Django's Model type and syntax, than we had at that time.
But using this approach was not in vain, since we used the simpler database generating from this method, to establish a connection between Postgres and Django, which made it much easier to continue with the second approach.
We can print the SQL that django generates, and it becomes clear, that our Mainteam model didnt have constraints or other relational features.

```
BEGIN;
--
-- Create model Mainteam
--
CREATE TABLE "mainteam" ("teamid" serial NOT NULL PRIMARY KEY, "name" varchar(50) NOT NULL, "description" varchar(600) NOT NULL);
--
-- Create model Membership
--
CREATE TABLE "membership" ("id" serial NOT NULL PRIMARY KEY);
--
-- Create model Teammember
--
CREATE TABLE "teammember" ("id" serial NOT NULL PRIMARY KEY, "description" varchar(200) NULL, "address" varchar(40) NULL, "age" integer NULL, "user_id" integer NULL UNIQUE);
--
-- Add field member to membership
--
ALTER TABLE "membership" ADD COLUMN "member_id" integer NOT NULL;
--
-- Add field team to membership
--
ALTER TABLE "membership" ADD COLUMN "team_id" integer NOT NULL;
--
-- Add field members to mainteam
--
```

Figure 7: Django-generated sql for the mainteam table

Postgres → Django
This was our second and final approach, and it was based on manually rewriting the Mysql to Postgres, and then from the resulting relational database, get Django to load the database tables as Django models. This removed the human error, that wouldve been caused by our

lack of knowlegde about Django models.
This approach was the better solution for us, since we didn't have to be experts in the Django Model, but instead had to have some experience with Postgres syntax instead. And based on our prior knowlegde, writing correct Postgres was an easier task, than mastering relational attributes in the Django Model.

Below is a Django Model, it consists of 3 parts. The model is a class, and each class equals one table in the database. The data fields, which are generated based on the data fields found in the Postgres database. And a class meta, which tells Django whether or not Django should "manage" the database table, and the name of said table in the database.
(It's also in the class Meta, that many of the relation attributes and constraints are set).

```
1 class Perscategory(models.Model):
2     teamid = models.IntegerField(blank=True, null=True)
3     category = models.IntegerField(blank=True, null=True)
4     catname = models.CharField(max_length=32, blank=True,
        null=True)
5
6     class Meta:
7         managed = True
8         db_table = 'perscategory'
9         unique_together = (('teamid', 'category'),)
```

8.2 Second sprint:

8.2.1 Vision

When planning for the second sprint, we once again looked upon our user stories to pick out the most relevant. We decided upon the 'account management' user story category, which is listed in the appendix.
All these user stories were related to the function of a user and the function of a team on the site. Fx. Creating and editing a user profile, creating and editing a teams site, applying to a team etc.
We were aiming to be so far in implementation that our customer could do a test session on the site, where he would use the site and test tryout the functions for half an hour during our meeting.

8.2.2 Implementation

Last sprint we set up our environment, which made it possible to start working on the actual implementation and features of the project. To get us started we picked out some user stories that we thought feasible for the sprint, and relevant for the project. They were all "frontend-focused" and were all depending on the Django "MVT" pattern, almost all of our user stories required;

- a database model to store and retrieve data from
- a view which is where most of the heavylifting in Django happens.
- a template which is the html and frontend of the project

The Models
For the user stories we primarily worked on 2 models, the Team(later Teammember) and the Mainteam model. (See appendix 7.1)
We quickly realized that the model had many fields that weren't necessary for the user stories currently in the scope, but at the start we did not realize the problems, that this would cause later on. The only fields of the Team model we were using at the start, was the name and the description. The first user story we were trying to implement was us01(creating and editing a profile).

The Templates
The template is what the user gets to see and work with, it handles everything html, ranging from simple html tags to forms with get and post requests. But it's also more than just html, Django templates can use a tag to b e parsed as Django functions instead of html. As an example, a function we've used many times is; "% if user.is_authenticated %", which means the content inside of this tag is only available to an authenticated user. This is applied on multiple things, including our navigation bar, where the options to go to your profile and your team is only available if you're logged in.
The navigation bar on the team template;

```
<ul class="nav navbar-nav">
    <li><a href="/">Hjem</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Gallery</a></li>
    <li><a href="#">Contact</a></li>
    {% if user.is_authenticated %}
      <li><a href="/profile">My profile</a></li>
      <li class="active"><a href="/team">My team</a></li>
    {% endif %}
</ul>
```

The Views
A Django view is like mentioned before, where the "heavylifting" of Django is made. This is where the functions that controls signals, input/output, etc are written
There are two ways to write a Django view;

```
class ExampleClass(View):
    def get(self, request):
        # Code block for GET
          request


    def post(self, request):
        # Code block for POST
          request
```

class-based

```
def ExampleFunction(request):
    if request.method == '
       POST':
        # Code block for POST
          request

    else:
        # Code block for GET
          request
```

function-based

They both have their pros and cons, and thus are good for different purposes. They aren't exclusive in the way that you can only have one or the other, so if you have a problem you think a function-based view is better for, but have class-based views for other features, you can mix and match as you'd like.

Conclusion of the sprint
When we were about halfway through the sprint, we had gotten ourselves quite stuck in our development, we had hit a spike. And we realized, we would probably have to re-write some of our earlier design. This realization happened while we were designing our forms making the user able to edit and change information on his profile and team. The problem was that we had just copied the database specifications from the old project, which meant a lot of the fields in the model was either useless for the current implementation or simply redundant. The first temporary solution to the problem was to simply set null=True on all these values, which meant we could continue for a while. But this wasn't solving the problem it was just hiding it until it resurfaced again.
All in all this meant for a not very successful sprint, we didn't get any of our user stories fully implemented, but it was a great learning experience, and we were hopeful that re-writing parts of the implementation would pave the way for the next iteration.

8.3 Third sprint:

8.3.1 Vision

We had a few goals for this sprint, but the priority was on getting the user stories from last sprint implemented, now that we had the new "foundation" of models to build on.

We had planned for the first part of the sprint to be spent on making what we had already developed, work the same way it used to, but this time with the new models. The models are essential to how a Django-app functions, and therefore this was critical to get right.
We also wanted to get started on the team page, which was originally planned for last sprint, but was delayed due to our unforeseen problems.
However it was not all about repeating the last sprint, we also wanted to implement something new, and for this sprint, "new" was the calendar.

8.3.2 Implementation

Starting fresh
Our first sprint was migrating the database to Django, and this was partly undoing that operation. To not lose the previous work, we saved the full migration in a document that is not part of the current implementation, and started working on completely fresh models. The idea was that instead of making models that was 100% like the old database, we would instead make models that were made with the functionality of the product in mind. The result of this was that we implemented parts of the different tables as we went on with the implementation. For example we didn't include the description field of a teammember, before we were working on the description functions of the website.

The new user page
We decided to go split the profile page into two pieces, a profile page which shows the logged in users data, and then an edit_profile page. As of now the current state of the page, is that it only includes 3 fields of the users profile, a description, an address and the age of the user. Both the profile page that shows the user profile, and the edit page can be seen in the appendix.

If we compare the new model to the old, both of which can be seen in the appendix, the newer one is much smaller, and like we described earlier, only contains the fields that we are currently using in the implementation. The pros and cons of this, is that it's much easier to work on the implementation, since we only have to worry about what we are currently working on. The downside is, that down the road, there might be a database constraint or another feature that will the difficult to implement.

The team page
A major component of the site is the team page, which has multiple functions, of which we have only implemented some so far. We were originally planning to work on this page in sprint 3, but because of the issues that came up, we were unable to get properly started on it. This meant that sprint 4 was the first time we really started putting in work on the team page. Out of its many functionalities we wanted to implement two things, we wanted to make it possible to establish a link between a user and a team, and we wanted to show which members were on the users teams. To establish this link we made a model to handle memberships between a teammember and a team, this model looked like so;

```
class Membership(models.Model):
    member = models.ForeignKey(Teammember, on_delete=models.
        CASCADE)
    team = models.ForeignKey(Mainteam, on_delete=models.
        CASCADE)

    class Meta:
        managed = True
        db_table = 'membership'
        unique_together = (('member', 'team'),)
```

We now had all the components for building a simple team page, so the only thing left was how to represent it on the website. Our current way of getting the current session is based on the logged in user. And what we wanted for the team page, was to show the teams that the user is part of, and the members of those teams.
To do so, we grabbed the logged in users teammember object, and then filtered all mainteam objects for mainteams where the user was a member. This was made in `views.py` and then passed onto the html template for the page.

```
def get(self, request):
    current_user = request.user.teammember
    current_teams = Mainteam.objects.filter(members =
        current_user)
    form = TeamForm()
    args = {'form': form, 'current_user': current_user, '
        current_teams': current_teams}
    return render(request, self.template_name, args)
```

Like described before, a class based view will have a get and post function, and this is the get function of the teampage view.
It gets the current user based on who sends the request, and from this "request.user" it gets the appropriate teammember and mainteam.
"args" is the argument we want to pass onto the html template, written as a python dictionary. And to pass them on, they get included when the get-function returns.
Both the relevant part of the html template and a screenshot of the team page can be found in the appendix.

Calendar

Lastly we wanted to get the calender in place. Our first idea was to create it in HTML and then have a javascript for functionality, however we learned that this solution would take longer and be much more difficult than what we expected. Therefore we ended up turning the calendar user story into a spike, and after evaluating the spike with a spike

test, we realized that embedding a public Google calender on the website was a much more consistent and easier solution, during the spike's test we further investigated the source code of the calender on the old website, and we found a PHP file containing a integrated Google calender. Which meant that this was possibly also the solution that was used on the old website. Unfortunately we do not know if this was an old or a new solution by Hans Jakob Simonesen, but either way, it was the solution we chose to bring to our 'customer' for the next meeting.

8.4 Fourth sprint:

8.4.1 Vision

We knew that this sprint would have to be spent writing more documentation than it's predecessors, as our previous priorities had been skewered in favor of developing and designing, rather than documenting.
With that said, we still had some user stories and features to fix and some to implement. These included further work on the team pages, and adding the new diary and blogging feature and general bug-fixing as well as finishing the design.

8.4.2 Implementation

Team page
One of the features we were missing with the team pages from our last sprint, was the feature of giving each team its own individual page. We did not quite get around to fully implement this feature, we were able to index the pages by the teams "teamid" but unable to fully apply these new indexes.
One might ask, how we would know that the pages were indexed, if we are unable to use them in the framework. But we know that these indexed pages exist since you can manually visit them by typing in a teams url in your browser. However we were unable to make the Django url-handler make links between these pages, fx. when redirecting from one html page to another.

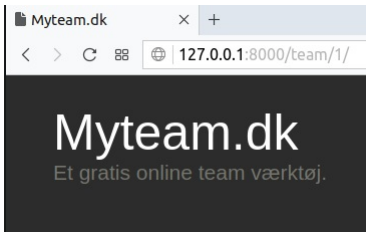


Figure 8: Team url example '127.0.0.1:8000/team/1/'

Diary and blogging
Due to a limitation on time, the diary and blogging features are not fully completed. They are partly implemented, and it's possible for a user to make a blog post. Likewise it's also possible for us, the developers, to choose how we want to sort these blog posts, since we have the data in the database. Our current blogging solution is a list of all the blog posts, sorted

by date created, and this criteria for sorting can easily be modified.
The reason that the feature is only partly implemented, is that there currently only exists blogs linked to the author of the post, but we would also like to have team based diaries. What this means is, that instead of having blogs be only attached to their author, which is the current solution, we would also like them to be attachable to a team as well.

Looking ahead
We were unable to reach all of the goals we originally set out to do. But we have halted the project at a point, where it's easy to pick up and expand upon. Either by us, or by a new team of developers. In case the project will be picked up by another team, we've documented both our work and compiled our list of user stories, to make it easier to complete the work we were unable to.

9 Conclusion

Our goal with this thesis was to gain an insight into agile software development by getting experience with the agile methodologies, through work on the *"MyTeam.dk"* project. In this process we also set out some goals for ourselves. We've described these as "main issues" in section 3.
One of these main issues was; "What are the dynamics for a small agile development team?" When we reflected on how working as a small agile team worked out for us, it became clear that some agile tools were good and some were not.
As an example, we would've liked to use more elements of Scrum, however we had to acknowledge that Scrum is better suited for bigger teams, and therefore most of our knowledge of Scrum comes from books and not from trial on the MyTeam.dk project.
On the contrary to Scrum, we found that Extreme Programming was a good fit for a small team, and we got to try many of the practices on the project. For instance we felt that pair programming was a great method for the more difficult tasks we faced.

We also found that software development can be a difficult process when first attempting it; but if you follow a methodology, you notice how some things go easier or faster than anticipated. This became clear for us, especially in the stages of our planning. The rules given by the methodologies meant that we, from the beginning, had an idea of how our project would progress.
Having the project broken down into user stories, also meant that we were able to split the work easier and evenly between the developers. Since user stories are by definition fleshed out and well described, the developer could start working on them immediately.

To dive further into one of the other main issues we set out to answer; "How spikes are handled in agile development?"
A spike is in many cases a user story that has not been expanded enough, which during development can become a problem if the story has been severely underestimated - which we discovered ourselves during our project. The most important aspect is to not stop working because of the spike, but instead get as many parts of the spike done as possible. This is

because the spike will be split into multiple smaller user stories, and to stay on schedule, an agile team wants as many of these new user stories to be implemented as originally planned. It will also be much easier for the team to analyze and split the spike into smaller bits, after having already spent time working on it. The parts that managed to get implemented is split into user stories that was done during the sprint, and the remaining parts that did not get done in the current sprint, gets planned for the next sprint, so the issue finally can get fully resolved and implemented. We studied and tried firsthand to work with spikes in an agile development project, and what we found was that spikes does not necessarily halt development, but rather require a more in-depth analysis of the problem. To achieve this level of higher understanding of a problem, you need to have spent time working on it. This was how spikes could go unnoticed during our planning. As an example we didn't have much experience with databases, and thus we believed that migrating the database from one DDL to another would be the hardest database-related task. But it turned out to be maintaining the design of the relational database, while implementing new features on it, that took us the longest time. At the time of planning the second sprint, we didn't even consider it a spike, simply because we didn't have the required knowledge.

We managed to keep the project moving, by being transparent with the customer and working on the parts of the problem we were able to. Which meant that when we split the spike up into smaller user stories for the next sprint, we had already made a lot of progress. Essentially turning one big unsolved problem, into smaller problems, of which some had already been solved.

Looking at our last main issue, we set out to implement a new interface on the website using the Django framework, and simultaneously synchronize said interface with a relational database. What we found was that Django is suprisingly well designed for this exact purpose. This originates in the fact that Postgres is a DDL which is partly designed for relational database functionality, and relational databases can be hard to stay on top of. For us to have a framework that represents the database in a more readable manner turned out to be a good decision.

This meant that we did not need to spend too much of our time working on the synchronization between the framework and the database. This decision turned out to be a blessing for us, since managing the database itself turned out to be an issue. We heavily underestimated the amount of constraints and specifications in the relational database, and therefore maintaining the structure of the database while working on the project was a difficult task. All in all it was a good learning process, and gave us a deeper understanding of the project managing aspect, and also from the eyes of the developer.

References

[1] Mike Cohn. (2004). 1st Edition.
User Stories Applied. Addison-Wesley Professional.

[2] Kent Bechs. (2004). 2nd edition.
Xtreme Programming explained: Embrace Change. Addison-Wesley Professional.

[3] William C. Wake. (2000). 1st Edition.
Extreme Programming Explored. Addison-Wesley Professional.

[4] Jeff Sutherland. (2014).
The Art Of Doing Twice The Work In Half The Time. Crown Business

[5] Alistair Cockburn. (2001). 1st Edition.
Agile Software Development. The cooperative game. Addison-Wesley Professional.

[6] Addison-Wesley Professional, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. (1995). 1st Edition.
Design patterns: elements of reusable object-oriented software. Addison-Wesley. Professional.

[7] Henrik Kniberg. (2007). 2nd Edition.
Scrum and XP from the Trenches. infoQ.com

[8] Thomas Broby Nielsen, Páll Skírnir Magnússon and Micheal Holm. (June 16, 2017). (2007).
Brugerhistorier indenfor projektstyring af spiludvikling. Bachelor Rapport DIKU for Projekt Luna 2017.

[9] *Manifesto for Agile Software, 2001. (Accessed 05-04-2018)*
http://agilemanifesto.org/iso/en/manifesto.html

[10] *Django overview, 2017. (Accessed 18-04-2018)*
https://djangobook.com/tutorials/django-overview/

[11] *Beretning om det digitale tinglysningsprojekt, 2010. (Accessed 21-05-2018)*
http://www.rigsrevisionen.dk/publikationer/2010/142009/a413-10/

[12] *Wikipedia page of extreme programming, 2018. (Accessed 29-05-2018)*
`://en.wikipedia.org/wiki/Extreme_programming`

[13] *Medium site with the scrum framework, 2018. (Accessed 28-05-2018)*
`https://medium.com/i-want-to-be-a-product-manager-when-i-grow-up/the-scrum-framework-95b16bc216c4`

[14] *Official documentation of the django template language*
https://docs.djangoproject.com/en/2.0/ref/templates/language/

10 Appendix

10.1 2nd sprint

10.1.1 Mainteam model

```
class Mainteam(models.Model):
    teamid = models.AutoField(primary_key=True)
    teamurlname = models.CharField(unique=True, max_length
       =64)
    teamname = models.CharField(max_length=128)
    sport = models.CharField(max_length=128, blank=True, null
       =True)
    nopers = models.IntegerField()
    maxweight = models.FloatField(blank=True, null=True)
    created = models.DateTimeField(blank=True, null=True)
    hits = models.IntegerField(blank=True, null=True)
    templateid = models.IntegerField(blank=True, null=True)
    color1 = models.CharField(max_length=7, blank=True, null=
       True)
    color2 = models.CharField(max_length=7, blank=True, null=
       True)
    color3 = models.CharField(max_length=7, blank=True, null=
       True)
    color4 = models.CharField(max_length=7, blank=True, null=
       True)
    color5 = models.CharField(max_length=7, blank=True, null=
       True)
    color6 = models.CharField(max_length=7, blank=True, null=
       True)
    color7 = models.CharField(max_length=7)
    color8 = models.CharField(max_length=7)
    topimg = models.BinaryField(blank=True, null=True)
    smsreminder = models.SmallIntegerField()
    mailreminder = models.SmallIntegerField()
    statereminder = models.SmallIntegerField()
    sendsmsok = models.SmallIntegerField()
    maximagesize = models.IntegerField()

    class Meta:
        managed = False
        db_table = 'mainteam'
```

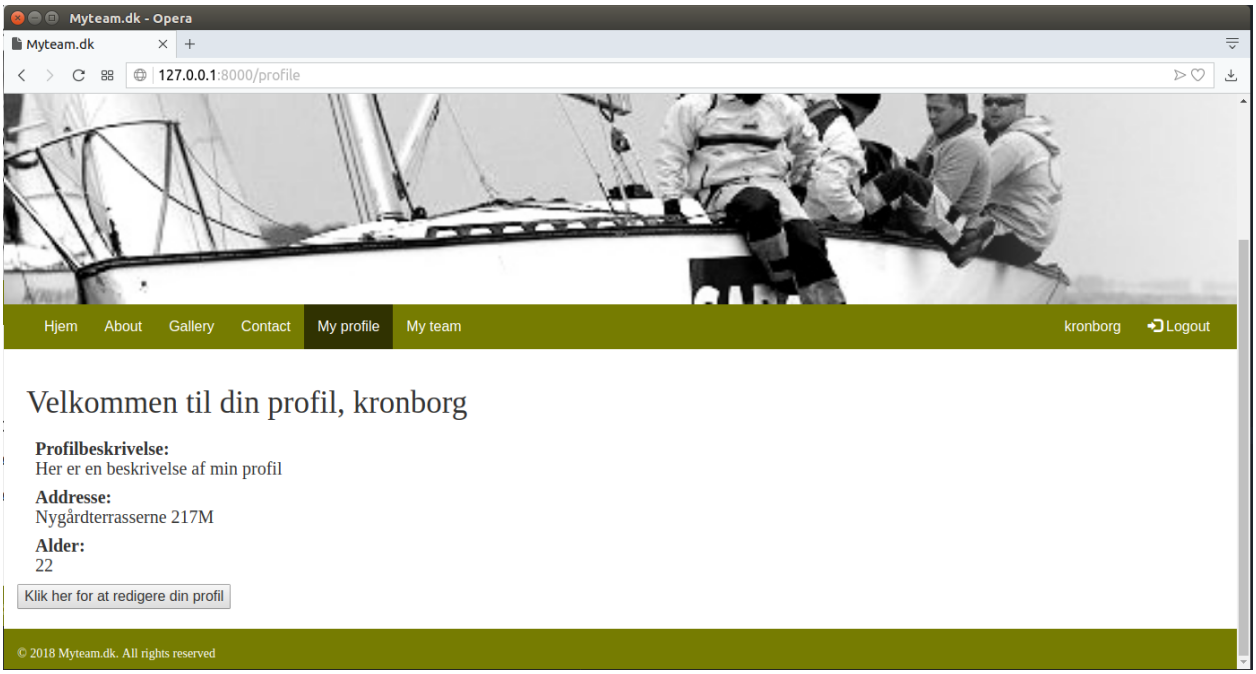
10.1.2 Team model

```
class Team(models.Model):
    persid = models.AutoField(primary_key=True)
    admin = models.SmallIntegerField(blank=True, null=True)
    admineditok = models.SmallIntegerField(blank=True, null=
       True)
    category = models.IntegerField(blank=True, null=True)
    active = models.SmallIntegerField(blank=True, null=True)
    passwd = models.CharField(max_length=150, blank=True,
       null=True)
    name = models.CharField(max_length=150)
    mobile = models.IntegerField(blank=True, null=True)
    tele = models.IntegerField(blank=True, null=True)
    worktele = models.IntegerField(blank=True, null=True)
    street = models.CharField(max_length=150, blank=True,
       null=True)
    postcode = models.IntegerField(blank=True, null=True)
    city = models.CharField(max_length=150, blank=True, null=
       True)
    email = models.CharField(unique=True, max_length=150,
       blank=True, null=True)
    publicinfo = models.SmallIntegerField()
    comment = models.TextField(blank=True, null=True)
    imgdata = models.BinaryField(blank=True, null=True)
    thumbdata = models.BinaryField(blank=True, null=True)
    position = models.CharField(max_length=128)
    description = models.TextField()
    birthday = models.IntegerField()
    club = models.CharField(max_length=128)
    proffesion = models.CharField(max_length=128)
    history = models.TextField()
    account = models.CharField(max_length=32)
    created = models.DateTimeField()
    sex = models.CharField(max_length=8)
    level = models.IntegerField()
    weight = models.FloatField()
    lastlogin = models.DateTimeField()

    class Meta:
        managed = False
        db_table = 'team'
```

10.2 3rd sprint

10.2.1 Screenshot of profile page



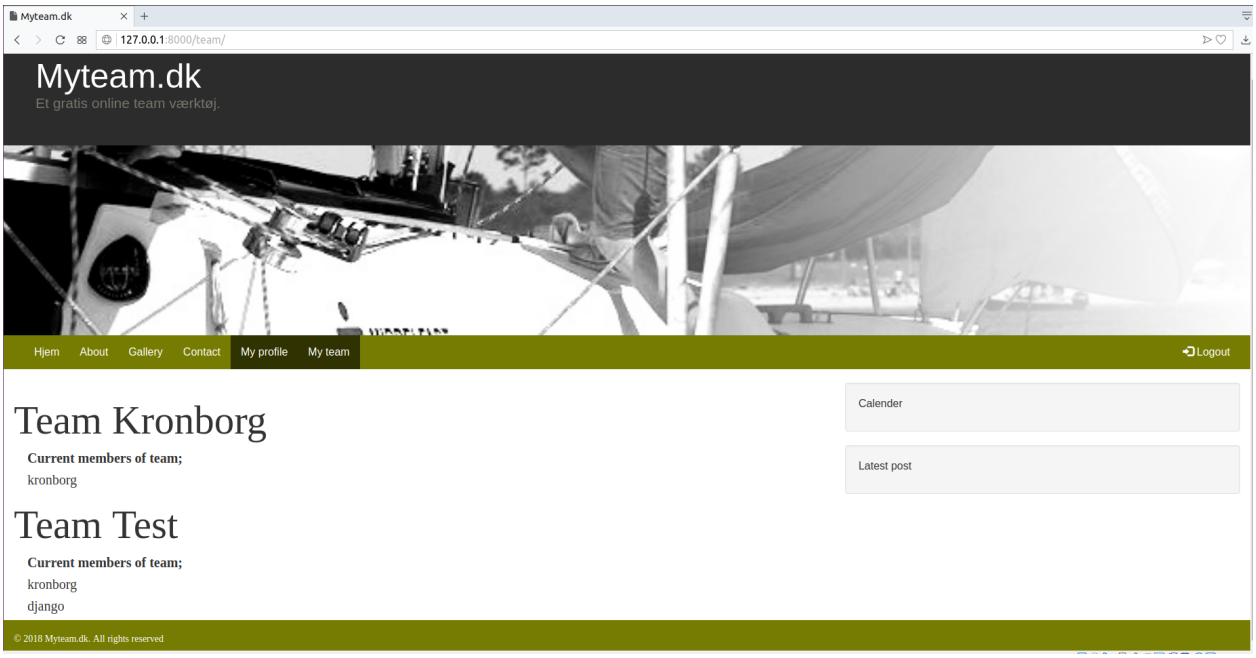
10.2.2 Teammember model

```
1 class Teammember(models.Model):
2     user = models.OneToOneField(User, on_delete=models.
         CASCADE, null=True)
3     description = models.CharField(max_length=200, default=''
         , null=True)
4     address = models.CharField(max_length=40, null=True)
5     age = models.IntegerField(null=True)
6
7     class Meta:
8         managed = True
9         db_table = 'teammember'
```

10.2.3 Teampage html template

```
{% extends 'base.html' %}
{% block title %}Frontpage{% endblock %}
{% block content %}
    <div class="col-sm-8 text-left">
      {% if user.is_authenticated %}
        {% if current_teams.all %}
          {% for team in current_teams %}
            <h1>{{ team }}</h1>
            <h6><b>Current members of team;</b></h6>
            {% for member in team.members.all %}
              <h6>{{ member }}</h6>
            {% endfor %}
          {% endfor %}
        {% else %}
          <p>You appear to not be part of any team</p>
        {% endif %}
      {% else %}
        <h1>You have to be logged in to see or edit your team
           </h1>
      {% endif %}
    </div>
{% endblock %}
```

10.2.4 Screenshot of the teampage while being logged in as user 'Kronborg'



10.3 Manifesto for Agile Software Development:

1. **Individuals and interactions, Over Processes and Tools:**

   The point of the first value of the manifest is to value people, more than processes or tools, because if the process drives development it's likely that the development team will be less responsive, which might cause the team not to meet the customer needs.

   This value can be narrowed down to a focus on communication, for when you are working with individuals communication is a dynamic progress, that happens when a need arises, meanwhile if the focus was on processes or tools, communication would likely be scheduled and focus on specific topics.

2. **Working Software, Over Comprehensive Documentation:**

   The second value, is a counterpart to the traditional software development methodologies which had a focus on a detailed documentation process to handle each phase of the project, however these list were to extensive and is/were often the reason for delays.

   However this doesn't completely remove documentation, it instead streamlines it in a form that gives the developer what is needed to do the work without being hindered by precise details. This is done by using user stories which have enough information for a developer to start working on a function.
   To conclude this value does value documentation, however it values working software more.

3. **Customer Collaboration, Over Contract Negotiation**

   The third value, is all about negotiation to accomplish a successful development process , it's important to understand the expectations of the customer. This is also a step in the traditional software development methodologies, however the difference is that it had a focus on defining and signing those requirements, at the start of the project.

   In Agile they still make use of contacts, but instead the contract is signed with the purpose to work with the customer and is based on a collaborative communication. This means that the customer is involved in the process of development making it easier to meet their needs.

4. **Responding to Change, Over Following a Plan**

   Traditional software development view changes as an expense, which is why they had such a focus on detailed requirements. However with Agile development the focus is to continuously delivering value to the customer through iterative development. This is possible since the time used on a iteration is shorter and therefor priorities can be shifted from iteration to iteration, and new features can also be added into one.

   This value embraces changes, for it allows the customer to get the product that he really needs, and this helps us achieving greater collaboration with the customer.

10.4 The twelve principles:

To make it even more clear what it is to be agile software development, these four values also derived the twelve principles, which further described the four values and were created as guiding principles for the four methodologies. [9]:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software

   The goal of this principle is to give the customer working software at regular intervals, this code can then be used to evaluate and respond to changing and evolving user requirements.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage

   In agile a customer can always come with changes to the requirement, the ability to avoid delays when a requirement or feature changes, is done best by spending more time with the customer to understand the change in the first place.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale

   Here is a focus on providing immediate value to the customers by bringing them working features. Each iteration or sprint should bring a working product to the customer.

4. Business people and developers must work together daily throughout the project

   For this principle there is a emphasis that there should be a engagement between between the different participant of the team and these should be collaborating in all the phases of a project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done

   Motivated teams are more likely to deliver their best work than unhappy teams, how to do this is given in the principal itself by "Give them the environment and support they need, and trust them to get the job done."

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation

   This principle emphasis is on efficient and effective communication over status meetings, it says that the most effective method of conveying in by face to face conversations.

7. Working software is the primary measure of progress

   The most important measure is delivering functional software to the customer. Customer satisfaction is what should be defined as working software.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely

   The agile team establish a sustainability and scalability speed at which they can deliver working software, and they repeat it with each sprint.

9. Continuous attention to technical excellence and good design enhances agility

   To gain agility, the team should use the right skills and a good design since this ensures the team can maintain the pace, constantly improve the product, and sustain change.

10. Simplicity–the art of maximizing the amount of work not done–is essential

    The team should try to identity the most simple solution, so time isn't used on a unnecessary solution, you should develop just enough to get the task done.

11. The best architectures, requirements, and designs emerge from self-organizing teams

    Teams that are self organizing deliver quality products.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

    With reflections over the development the team is able to improve and thereby improve skills, and techniques which makes it easier for the team to complete the next sprint .

10.5 List of User stories

Account management:

- As a user, I can create and edit my profile, so that i can customize my experience.
  Test: We can create a profile, and can change customization options of said profile.
- As a user, I can create and edit my profile, so i can position myself as a sailor.
  Test: We can create a profile, and can change the sailor position of said profile.
- As a user, I can apply to a team, so that i can participate in team planning.
  Test: With a user account, we can send an application to a team.
- As a user, I can be part of several teams, so that i can participate in multiple team plannings independently.
  Test: Users can be in several teams and get relevant information from the different teams. Example: User is part of 2 teams, user can participate in an event on behalf of team 1, but also request reimbursement only from team 2.
- As a user, I can indicate which dates i am available to an event, so that event planning is clear.
  Test: Users can see events that are listed conversely chronologically, the user can then mark which dates they are available, so events can plan accordingly.

Reimbursement of expenses:

- As a user, I can pay bills to my team captain through the website, so that we don't have too manage payments while sailing.
- As a user, I can bill the team for things i've covered for, so that i can get reimbursement.
- As a user, I can review my billing history, so that i can get a total account for my expenses.
- As a user i can hand out the expenses to the related users, so that users pay their part.
- As a user i can ask the system to implement a reimbursement plan, so that users are notified.

Planning of events:

- As a team-admin, I can schedule a race event, so that the teams planning can be maintained.
- As a team-admin, I can cancel a race event, so that the teams planning can be maintained.
- As a team captain, I can schedule my team for events, so that the teams planning can be maintained.

- As a team captain, I can send out reminders to my team members, so they can get back with a response.
- As a user, I can receive a reminder for upcoming events, so i can get back with my response.

Posting and blogging:

- As a poster, I can decide whether posting is private or public, so that i can control who sees the posting.
- As a user, I can decide to release my public information, so that i can release it in a proper form.
- As a user, I can share social updates(blog) to the site, so that i can share my experiences with followers and competitors. (Private/Public content)
- As a user, I can write additions to the team diary, so that i can share my experience.

Crew assets:

- As a user, I can set my position, so that i can post my qualification.
- As a user, I can view my teams ranking, so that i keep up with the ranking.
- As a team admin, I can list my sponsors, so that they can be promoted within the team.
- As a team admin, I am able to request a substitute, if my team is incomplete. So that my team can still participate in events.

Administration:

- As a Sysadmin, I can manage user profiles, so that i can delete unwanted or outdated team and team member profiles and related data.
- As a Sysadmin, I can reset user passwords, so that users can regain access after having been locked out.
- As a Sysadmin, I can grant and remove access to parts of the website, so that followers cannot perform team tasks, so that team users cannot perform team admin tasks.
- As a Sysadmin, I can schedule downtime, so that people can plan accordingly.
- As a Sysadmin, I can edit the website, so that i can decide the default user experience.