

System Development Report

OEADM16EIC Group D

Third Semester Exam Project



Roxana Ion



Hedviga Gerina



Jonas Laursen



Matthew Peterson

Table of Contents

Introduction	4
Problem Description	4
Solution Description	5
Tools used	5
GitHub	5
GitHub Board	6
Google Drive	6
Start Up	6
System Development Methodology	6
Whole Team	8
Planning game	8
User Story Prioritization	9
System Metaphor	10
Dropped the idea of Sprints	10
Requirements Gathering	13
Personas	13
User Stories	13
Rich picture	14
Prototype	15
High Level Design	16
Wireframes	16
SSD	17
Software Architecture	19
Design patterns	19
Low Level Design	22
Design Class Diagram	22
Development	23
Pair Programming	24
Hardware	24
Development Environment	24

Source Code Control	24
SVN vs GIT	25
No Source Control	25
Testing	25
Testing Tools	25
Test Driven Development	26
User Acceptance Tests	27
Deployment	27
Choosing Hosting Platform	28
Maintenance	29
Retrospective & Conclusion	30
Problems & Issues	30
What went well	31
Working without sprints	34

Introduction

The report is made as a part of participating in a project on the courses: System Development in the third semester of the Computer Science education at Erhvervsakademiet Lillebælt.

We have initiated contact with multiple companies, including our school to try and continue our previous room reservation project or rework it, but we did not get a usable answer from any of them. Then we also contacted the LærDansk Organization. We thought about them, as Hedviga is a student there and had various issues in mind.

We have chosen to work for LærDansk which is a school responsible for teaching international students/people danish language. We are going to try and solve the problem they are facing regarding their inefficiency of their website. Further description is provided in the Problem Description.

Problem Description

We have contacted LærDansk in regards to improving their web-based services. Today, they have a google site, a CMS-like system, to organize everything on the website. Hedviga, along with other students, find the google site difficult to navigate, as it is badly structured and not very user friendly, this makes it difficult for their students to find important documents.

On our initial contact with LærDansk, we suggested that they could benefit from a phone app. This app would allow students to have access to their time schedules, lecture plans, and potentially even do exercises. They were interested in this idea, so we agreed that we would try to solve this problem for them.

In our next meetings we discussed the problem more in-depth. It seems that students, volunteers and teachers are not using the provided google services to their full potential. The front page has too much information and navigating to the most important parts of the website is difficult.

This causes various problems including: lost information, confusion, frustration and in the end students avoid using the service at all. Consequently, students are not prepared for lessons, are not doing exercises, are missing events and are not receiving help from the teachers and other volunteers. This issue occurs all the time and needs to be fixed as soon as possible.

Solution Description

Our suggested solution is a phone app that interacts with our Web API. The Web API is going to be the core for all UIs. Right now our only UI is going to be the phone app, but we think that in the future LærDansk should, instead of using the google sites, use the Web API to create a full website and only display minimal information in the google services with references to where the rest of the information is or redesign the google sites as they see fit.

The phone app will also allow easier access from anywhere as long as the student has their phone with them, thus, this will most likely lead to students using the app instead of the google sites and they will only navigate on the website when they need extra information, want to check their emails or other, similar actions.

We will focus on making the phone app easy to use and navigate, containing various useful functionalities. Since this project is for our exam, it will be free of charge for the company, and this was another reason why LærDansk was interested in our project.

Another solution to solve the problem would be to reorganize the google site and maybe even make user training on how to use it, but then it would not have been possible to work with this problem for our exam project. Google sites are not very mobile friendly, and would not solve the issue of students not being easily able to find the information, when without their computers.

Tools used

GitHub

We used GitHub to support traceability because Github repositories allow us to create Issues. We will use Issues as our decision log when we work on the code, artifacts or anything else by linking a commit to an Issue. By committing and writing comments when we make changes, we will be able to see who made the change and someone else can do the quality check against our pre made criteria.

GitHub Board

We used the “Project” feature of GitHub to create a Kanban Board for our project to track our progress in the current sprint. After making a decision and creating an issue we link our GitHub project tasks to them. Projects also allow us to create backlog tasks from bug reports.

Google Drive

We used Drive for writing this Report and other documents/artifacts. It was very helpful for us to be able to work on a document all together and it greatly increased our productivity.

Alternatively, we could have used software such as OneDrive, but we did not feel Microsoft's “live” documents work as well as Google Docs.

FURPS+

The main reason why we decided to create FURPS+ is to prioritize our backlog, gather requirements and assure the quality of the project.

The main aspects we choose about FURPS+ are usability, functionality and physical constraints. These are the ones our P.O. considered would be the most important for our project based on the description we made about FURPS+ (see description in Appendix section 5).

FURPS+ was also useful in deciding on some of the requirements our system would have and, in the end, we were able to use it as quality criteria/goals for the program.

Start Up

We have started up this project by creating a group contract with various sets of rules that would make the working environment more pleasant and efficient. This contract helps us to be more organized with our duties, it supports the team members to be more competent and assure a higher quality of the teamwork. Some disadvantages of not having a group contract would be: unorganized teamwork, communication across the team might be less efficient or even minimize the expectations of each member. These are some of the risks we are trying to avoid in our group, hence the decision of creating a group contract. (See contract in Appendix 1)

During the Startup stage of our project we had meetings with our product owner to figure out, in detail, the problem we need to solve for them.

One of the requirements we had to drop was integration with UNI Login. LærDansk would have liked us to use UNI Login as authentication, but upon our initial research we found this requires a much larger business infrastructure to gain access to, such as CVR number and special servers, meaning we had to leave this to LærDansk. Our solution was to provide an interface system, where authenticated users can easily be injected into the system.

System Development Methodology

When the time came to make our SDM decisions, one of the first things we took into consideration was the Process Models. We started by making an analysis of each process model, with its advantages and disadvantages, to find a model that fits our project the best. Our arguments for and against the various process models can be seen in Appendix section 2.

We came to the conclusion that Agile would be the best process model for us, and this decision was influenced by the following factors. In agile, it is easy to add new features or ideas after the project has started, even in the later stages of development; it is relatively easy to drop or change features if the product owner decides to do this; as we work on sprints, the quality of the previous parts will be assured and there is a high degree of collaboration between the product owner and our team. We can develop simple solutions and improve and add features gradually.

Some of the disadvantages we have to deal with are: it is difficult to get an overall plan of the project, as things can easily change; time can be an issue, as we might not be able to complete everything we planned to and we need to have frequent meetings with the product owner.

Other alternatives we took into consideration were:

The Predictive process model, some of the advantages would be: we could have proper deadlines for our features and it has good documentation and traceability since everything is planned out in the beginning, but because LærDansk is unpredictable, this way of working could lead to serious problems in our project.

The Iterative process model, the advantages it comes with are: as we deliver a working, but not completed, program, our product owner can see everything working and decide what they want to improve or drop. But the disadvantages are: we might spend time on features that might be dropped and it is not easy to change or add ideas later in the project.

The Incremental process model, an advantage it has is the fact that we can add new modules after the project's start-up. The disadvantages are: a lot of work can go into a feature/module to only have it dropped in the end, we might end up with missing features and we cannot revise features once they are done.

We continued by looking through and studying all of the SDMs we have learned about in the lectures, to figure out which of the methodologies fits us the best.

We came to the conclusion that Extreme Programming will be the best approach for this project. The reason for this, is the fact that our product owner can change their mind about features and will be able to drop them easily, and this methodology supports this. XP satisfies our needs by being in the low-high uncertainty/high complexity. At this moment, we do not know exactly how complex this project will turn out to be, so XP will allow us to split it up and assure the quality of previous features, making the whole project simpler to work with.

From our own experience with real IT companies, we have observed that many of them use a variation of Extreme Programming. At the same time, we have done more research on this working method in our Video Project, thus we feel the most confident following the XP principles. Agile and XP work well together. These are three other reasons for our SDM choice.

Extreme Programming does not satisfy the needs of our curriculum, mainly because of the lack of documentation and traceability, hence we have decided to modify this approach by creating UML diagrams and other artifacts, also a decision log to better check against our quality criteria.

We are planning on using a GitHub Project to organise our tasks and other things that need to be done within the project. This way, we can better visualize our workflow and keep track of our status and progress.

The main alternative to Extreme Programming would be RAD. RAD is also a rapid methodology, that works in a more waterfall like structure. RAD works best for a quick prototype that is then scrapped later, where XP can work for full software as well.

We decided not to use RAD, mainly because it has even less documentation than XP, and we felt our project would become too big to follow RAD.

Whole Team

The “Whole Team” is a practice that Extreme Programming encourages. It makes sure that every member is part of the team, including the product owner, and that all of us are working towards a main goal, to create a more valuable product.

We decided to follow this practice so that each of us would make the project more efficient. Everyone is responsible for what is happening inside the project and no blame will be placed on anyone if something goes wrong.

As for our project, we decided that Hedviga will be the representative from the product owner’s side as part of the “whole team,” since she has been a student there for more than one year and encountered most of the issues the students at LærDansk have to deal with.

Planning game

The main planning process within extreme programming is called the Planning Game. This is used for gathering requirements, hence it should have been in the “Requirements Gathering” section, but since this is a practice of XP we decided to put it under the SDM section.

The Planning Game is divided in two stages:

Release Planning: This stage took place during the start-up phase of the project, where we had meetings with our product owner to figure out what the requirements of our project would be.

This stage consists of three phases:

Exploration Phase: The way XP is doing this is that the customer gives the requirements which will be written on user story cards. We did not use these cards, but electronic sheets where we would later transform them into Epic and Ready stories.

Commitment Phase: In XP this is the phase where the developers will commit themselves to the functionality that will be included and the date of the next release. During this phase our team did commit to some of the functionality our software will have but we did not set any release date because we decided we would not be doing it in small releases but in a big release at the end of the project. This is where the user story prioritization is done by the P.O. and us. See further description about the prioritization in the “User story prioritization” section.

Steering Phase: This is where we adjusted our plan, added, changed and removed new requirements during our later meetings and sprints.

Iteration/Sprint Planning: During this stage the P.O. was not involved, only the team. This is where we decided what our tasks are going to be.

This stage consists of three phases:

Exploration Phase: This is the phase where we translated our requirements/US into tasks.

Usually you put these into task cards but we decided to put these tasks on our GitHub Project.

Commitment Phase: During this phase we decided who would be responsible for each task, and usually you also give an estimation of the time for each task but we decided not to do this because we are not experienced enough to determine exactly how much time a task requires.

Steering Phase: In XP this is where we are going to start and finish the tasks, so it does not belong in the requirements gathering but in the other stages of the unified processes.

User Story Prioritization

In Extreme Programming there are a few ways to sort and prioritize the user story and for this project we tried to adopt these. Prioritization is done in the Release Planning stage under the Commitment phase.

FURPS+ had an influence on prioritizing the Epic user stories as well.

The first criterion of sorting is by value. This is where our P.O. decided which of the user stories are the most important and significant and which user story he would most like to have.

The next criterion of sorting is by risk. This is where our team sorted and decided which of the user stories have the least risk to implement. The way we did it is by checking the completeness and volatility of the US and how complex the user story is.

For example, we can show how we did it for one of our Epic user stories "US1 - As any User, I want to be able to interact with the lecture plan so that the class can be prepared"

(The numbers in the brackets show the amount of votes given)

We checked what is the completeness and how well we know the details of US1.

- Complete:(3)
- Incomplete:(0)
- Unknown:(1)

Then we checked on volatility, how likely is US1 going to change.

- Low:(2)
- Medium:(2)
- High: (0)

Then we checked on how complex US1 is and how hard it is to implement.

- Simple:(1)

- Standard:(3)
- Complex:(0)

Based on our and our product owner's decisions, we came to the conclusion that US1 will be the most important and valuable to implement.

System Metaphor

The System Metaphor makes it easier for the whole team, including the product owner, to understand and agree on how the system is working and how the problem and solution are perceived. A shared vocabulary is used for the metaphor story so everyone is familiar with the terms being used.

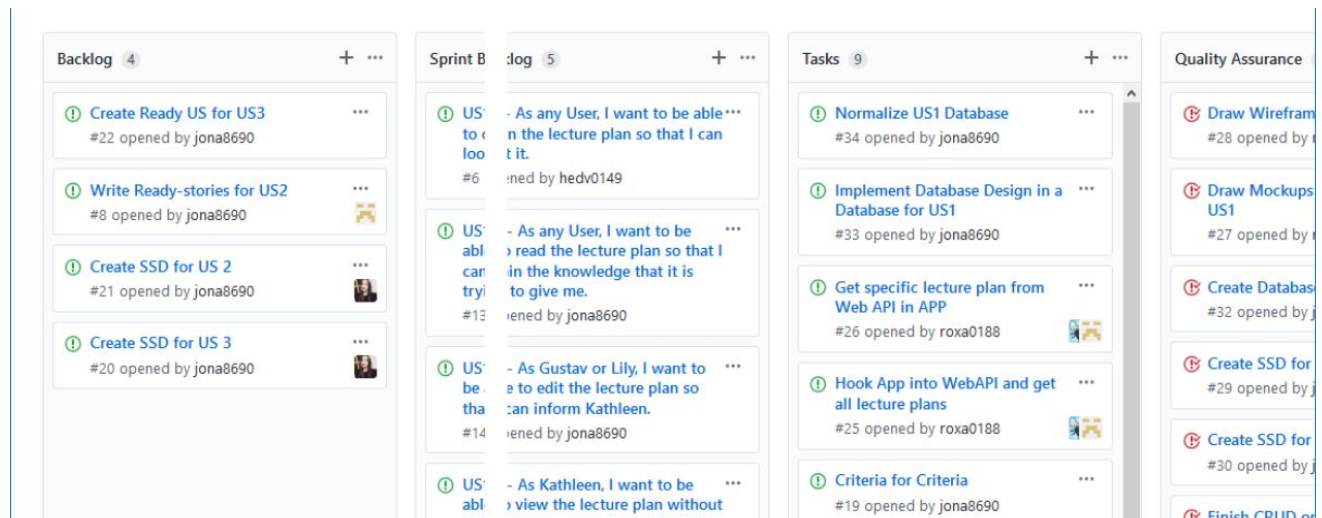
Despite the fact that the System Metaphor is one of the core practices of XP, we decided not to implement it, mainly because our product owner obtained the necessary information on how the system is going to work from our meetings and from our problem description, solution descriptions and from different types of artifacts; but we are aware of this XP practice.

Dropped the idea of Sprints

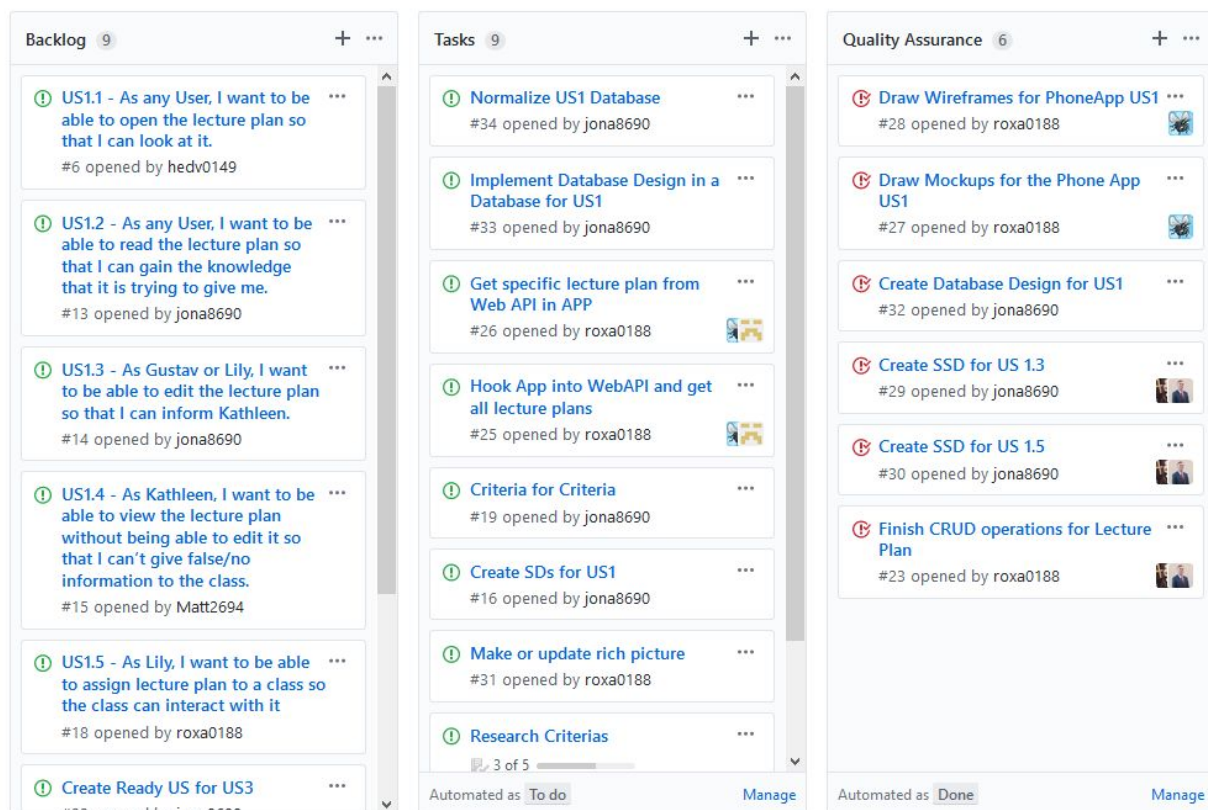
As discussed below in Requirements Gathering; User Stories, we had issues figuring out which Ready Stories to include in the sprints. This meant that we ended up having difficulties continuing our sprints. We felt the need to work on tasks outside the sprint very often, and continually changing the tasks in the sprint was very confusing.

Hence, we decided to drop the idea of forcing tasks to be confined within sprints. Our idea was to simply run down the tasks as necessary, giving us more freedom about which tasks to work on.

Kanban Board before:



Kanban Board after:



We still kept the Backlog organised by priority, and we still had retrospectives at times during the project, but rather than focus on the sprint itself, we focused on the tasks we had completed since the last retrospective.

Considering the goal of allowing us to move on quicker, we began to focus more on High and Low Level Designs, but we still worked on the code implementation.

Requirements Gathering

Personas

We created three personas to represent the different users of the program, one student, one teacher and one admin. We needed to create the personas before we could create the user stories and the personas helped us to create some of the requirements for our system.

We only created three personas, this meant that we did not get as many requirements as we could have had we made more personas, for example, different aged teachers and different education. This would have allowed us to see our program from different angles which could have changed the way we made it. It would have been more user friendly if we had done this.

An alternative to using personas could be if we just used actors. Actors serve a similar purpose to personas but they are used in use cases. They are not, however, as descriptive as personas and this would have caused us to lose the ability to get requirements from them.

Our personas can be found in the Appendix section 13.

User Stories

User stories are short, simple descriptions told from the perspective of the person who desires a new system. We decided not to create traditional user stories since they do not explain requirements in as much detail as, for example, use cases do, but our way does. The normal way of doing user stories is by using a template to describe requirements:

As a < type of user >, I want < some goal > so that < some reason >.

We put more effort into them than just filling out this template. We divided user stories into 2 groups - Epics and Ready. Epics are large user stories that are made directly from requirements, and they allow us to sketch out the requirements to see the overall idea.

The next step is to break it up into the Acceptance Criteria. The Acceptance Criteria is a list of smaller requirements that need to be met to complete the Epic. From this list, Ready stories are made. They are small user stories that the team can create tasks from.

By doing Epics and Ready user stories, the team gets a better understanding of what parts will be connected, the stories require approximately the same time and effort to complete and thus the tasks need, approximately, the same time to finish. A consequence of Epics and Ready stories was that we needed to spend more time on them than if we just created the traditional user stories, research, getting a complete understanding of them for the whole team, making criteria (appendix section 11.2) to check them against, creating them and finally checking them using the criteria.

Problems occurred when we started to use Ready stories in our sprints. If, for example, we put the Ready story for US1.2 into the sprint, but we had not yet made US 1.1, we would have problems implementing the code for this user story. Sometimes as a team we did not think about how closely connected Ready stories were, which made it hard for us to continue using sprints.

The user stories were quality assured by checking them against criteria we researched which can be found in Appendix section 11.2. All our user stories can be found in Appendix section 4.

We have inserted a section of our US as an example:

US1 - As any User, I want to be able to interact with the lecture plan so that the class can be prepared.(EPIC)

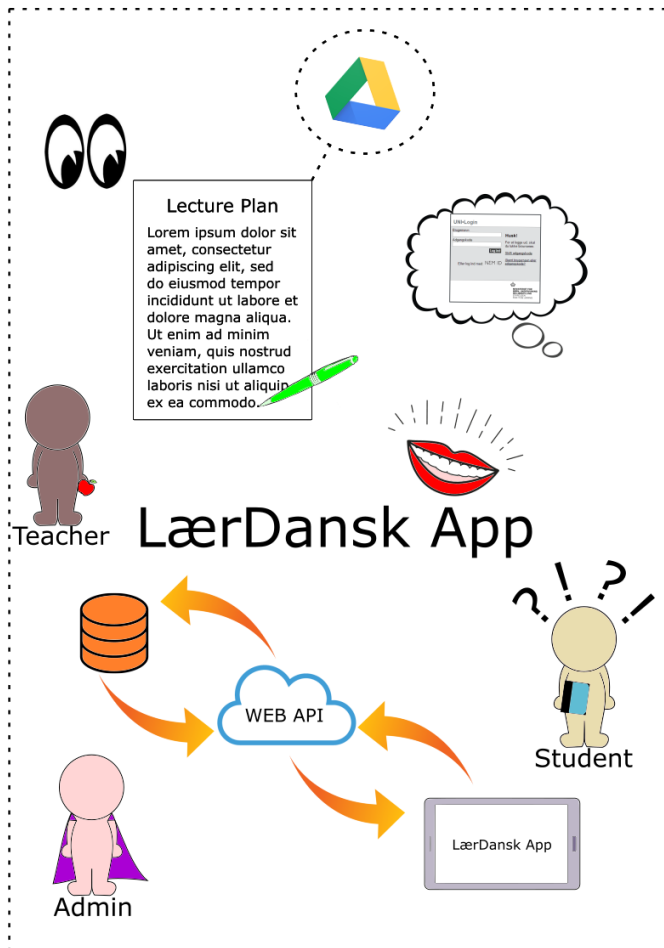
I can open the lecture plan. (ACCEPTANCE CRITERIA)

US1.1 - As any User, I want to be able to open the lecture plan so that I can look at it.(READY STORY)

I can read the lecture plan.(ACCEPTANCE CRITERIA)

US1.2 - As any User, I want to be able to read the lecture plan so that I can gain the knowledge that it is trying to give me. (READY STORY)

Rich picture



In one of the lessons we were introduced to the rich pictures. We were also advised to make one for our projects. We started by making the rich picture on a piece of paper but later, to add things and improve it, we had to redo it and we decided to use the Inkscape drawing program to create it. The drawings were more understandable in the new rich picture, and we showed it to our P.O. to help them understand what we wish to do.

Our rich picture shows that we have 3 types of users, all with different needs. In the middle we have the project itself - LærDansk App, as that is our solution for LærDansk's problem, below it we show how it will work and that WEB API is at the core of our solution, so they can use other UIs with it and not just our app. In the "thought bubble" we have the UNI Login interface, since they would need to implement that

themselves, as was mentioned earlier. The most important part is that the lecture plan can be edited and read. Then we can display it in the google sites that they already have. The final product must bring happiness and satisfaction to our P.O. and the users.

Prototype

We made the prototype so that our product owner and the development team could get a better idea of how the system would look and work. Prototypes are essentially mockups that have some of the functionality of the program implemented. This allows the person using the prototype to actually feel like they are using the system and they can get a much better idea of what they like and dislike about it. We made the prototype in a program called Pencil, because of this, we can not show the prototype, so it is shown as a mockup in Appendix section 8. We will show the prototype in use during our presentation.

Some other ways we could have accomplished this would be by creating wireframes or by creating mockups. We decided to make a prototype rather than a mockup because we like being able to actually click on the buttons and see how it works. This did require us to spend a bit more time on this step than if we had just stopped after doing the mockups since we had to do research on how to actually make the buttons work in the mockup creation program we were using.

We have placed the Prototype section in the Requirements Gathering part of the report even though we did the prototype after we made the wireframes. The reason for this is because the “Beginning Software Engineering¹” book had prototypes in the requirements gathering phase, and we did use it to gather some requirements, but in our opinion, it should be placed in the Low-Level Design because it was used mostly by the development team as a guide in the creation of the UI.

High Level Design

Wireframes



We made wireframes for different views because we had to show our product owner our idea/sketch of how we think the app should look, to satisfy their curiosity and help them get a better idea of what they want. It also helped us to see if other requirements are needed.

An alternative to a wireframe is a mockup or prototype, which we also made to be 100% sure that our goal would be reached. As we made them, our team and LærDansk had a meeting to discuss both the wireframes and prototype. After this meeting we started to work on the phone app using Xamarin.

The consequences of this decision were that we spent a lot of time making these artifacts as well as searching for criteria, checking

¹ Stephens, Rod. *Beginning Software Engineering*. John Wiley & Sons, 2015.

those, and then checking them against the criteria (See criteria for Wireframes in Appendix section 11.1.2).

A new way of achieving our goals could have been to include some sketches of our app in the rich picture to show Lærdansk how the app looks, but we might have missed some extra requirements and the idea would not be as clear as it is now.

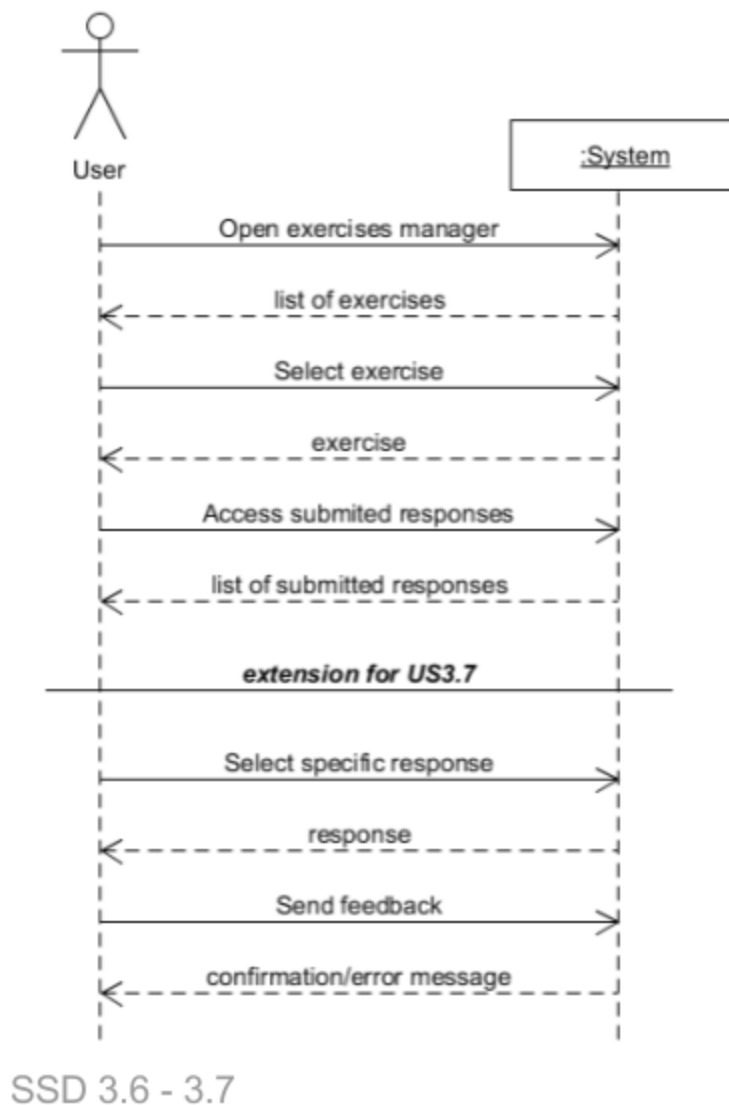
You can see more our wireframes in the Appendix section 7.

SSD

We drew SSDs in order to determine the successful flows, steps, inputs and outputs of our Ready user stories. Initially, we drew SSDs for the Epics, but they were of little help to us. They actually made us more confused because there were too many flows included.

The SSDs for the Epic stories were difficult to check against the criteria, seen in Appendix section 11.3. An exception to this was US2 because we did not create a proper login system since our product owner is using UniLogin. Instead we decided to make a generalized hard-coded user login, this is why it was possible for us to create an SSD for the Epic story US2, since it had the same flow but different type of users.

Since some of the Ready stories were too small, they did not require artifacts to explain the actor's interactions with the system. We decided, as a team, to make SSDs for Ready stories 1.5, 1.3 and 3.6 + 3.7 and a general SSD for US2.



Alternatively, we could have not made SSDs since they are not required by XP, but we decided to make them anyway. This was because Ready stories made the SSDs easy to draw, and if we had not made them, we would always need to remember the flow of the user story which could cause confusion among the team members or could even cause the program to be made incorrectly if the team members have slightly different recollections of how it should be made.

Our new way of doing SSDs was by combining 2 flows. In SSD US3.6 we added Ready story 3.7 as it makes sense to not only access the exercises but also to give feedback on them. The line that divides this SSD between the two Ready user stories is not mentioned in “Applying UML and Patterns²” by Craig Larman or anywhere else.

² Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed., Addison Wesley Professional, 2004.

A downside to this way of making SSDs is that if the two flows do not go together as well as it at first seemed, it could cause even more confusion.

You can see all SSDs in Appendix section 10.

Software Architecture

When we had to decide on our software architecture, multiple factors were taken into consideration. First, and most importantly, the exam requirements of being a web-based service. This does restrict our options a fair bit.

Initially we discussed 3 ways we could have set up this project. Firstly, we could have made a simple MVC website, single project. Having a simple, small project could have had advantages in aspects of documentation. Less code, less mess. Developing a website would also allow our application to instantly work on both computers and phones, however, to implement this kind of fluid design properly requires vast knowledge of CSS3 and HTML5.

In this aspect, we could have also simply made an HTML/JavaScript site using firebase as our backbone, but we felt this would make it difficult to reach our Web API goals.

Alternatively, we could have made the application purely in the Xamarin framework using the MVC pattern, but yet again, this would make it difficult to reach the goal of a Web API. Doing this would also lock us down to a pure phone app, limiting the ability to expand to other systems in the future.

Our solution, a merge between Xamarin and a Web API. We develop the Web API using Models and Controllers, and implement the view as Xamarin. The approach does increase our codebase, as we need proxy classes, and initiate calls to the API, but we feel the expandability of this approach outweighs these negatives.

We have created 2 solutions, the “Core” Web API Service, containing a DAL, Models, and unit testing, and a Xamarin solution, which simply executes call to the Web API and presents the view. This means that our system can also be natively implemented as a website, or in the future, other ways.

Design patterns

Application Storage

Of the ways to do application scoped storage, our favorite is by far the singleton. It is a well known pattern that ensures only singular instances of an object can be created, hence forcing that instance to be used everywhere.

We use the singleton pattern for our repositories, these contain information that should be shared across the application, such as the user, exercise, and lecture plan repositories. This, for example, ensures we cannot have multiple lists of users created.

Alternatively, we could have ensured the same repository instances gets injected via dependency injection into the classes, however this leaves room for slipups. Using a singleton simply avoided this, but also still allows the use dependency injection.

```
private ILecturePlanRepository RepoLecturePlans;

1 reference | Jonas Laursen, 21 days ago | 1 author, 1 change
public LecturePlanController(ILecturePlanRepository RepoLecturePlan)
{
    this.RepoLecturePlans = RepoLecturePlan;
}

0 references | Jonas Laursen, 21 days ago | 1 author, 1 change
public LecturePlanController()
{
    this.RepoLecturePlans = LecturePlanRepository.Instance;
}
```

Above is an example of our dependency injection. If the class is being instantiated where injection is given, the class will use this, otherwise get the instance from the repository. This allows us to easily instantiate versions of the class, without having to worry about getting the right instances, while still allowing for injection, making testing easier.

Our third option for persistent application scope would be to interact directly with the database. In a previous project we did this, where the repositories were simply acting as a facade to the DAL, saving and getting information and objects directly to/from the database. Facading the Database did work as intended, but was quite resource intensive, which is why we wish to avoid doing this again.

Here we could also have used an Object Relational Mapping framework, such as entity framework, but properly setting up and using these can take time and implies a more finished, stable application, not the rapid prototype we are trying to develop. Entity framework increases complication in making objects, as proxy classes have to be implemented using the entity framework tags. This reduces our ability to quickly create and drop features, but using the ORM would have solved our issue above, with dealing with object instantiation mentioned above, as the ORM would only get the necessary objects.

Using singletons was, for us, the fastest and most efficient way we could implement a application scope storage for this project.

Proxies

Most of the model classes in the core project have similar classes in the phone app, to act as remote proxies. They exist in order to encode and decode the models when being sent from and to the Web API.

Using a Proxy has helped us keep consistency between which classes are being used. This means that when talking together, there was less confusion when discussing the different projects (core & phone app). It did mean we had to specify which class was being referred to, e.g. "Does the User class in phone app do X function?", but we feel this is more than an appropriate payoff to ending up with different class names or structures.

If not using remote proxies, we could have just used JSON strings, and parsed these strings and edited them directly, but this would likely have ended up being more difficult and time consuming.

An alternative all together would have been to not use a Web API, but this was a requirement for the project, hence we could rule out solutions which did not use this. We talked about this during the software architecture section as well.

MVC/MVVM

We also utilised the MVC pattern and its extension MVVM to help structure our software in a more developer friendly way. This was crucial, in order to achieve the separation we wished, described more deeply in the section: *Software Architecture*.

Low Level Design

Design Class Diagram

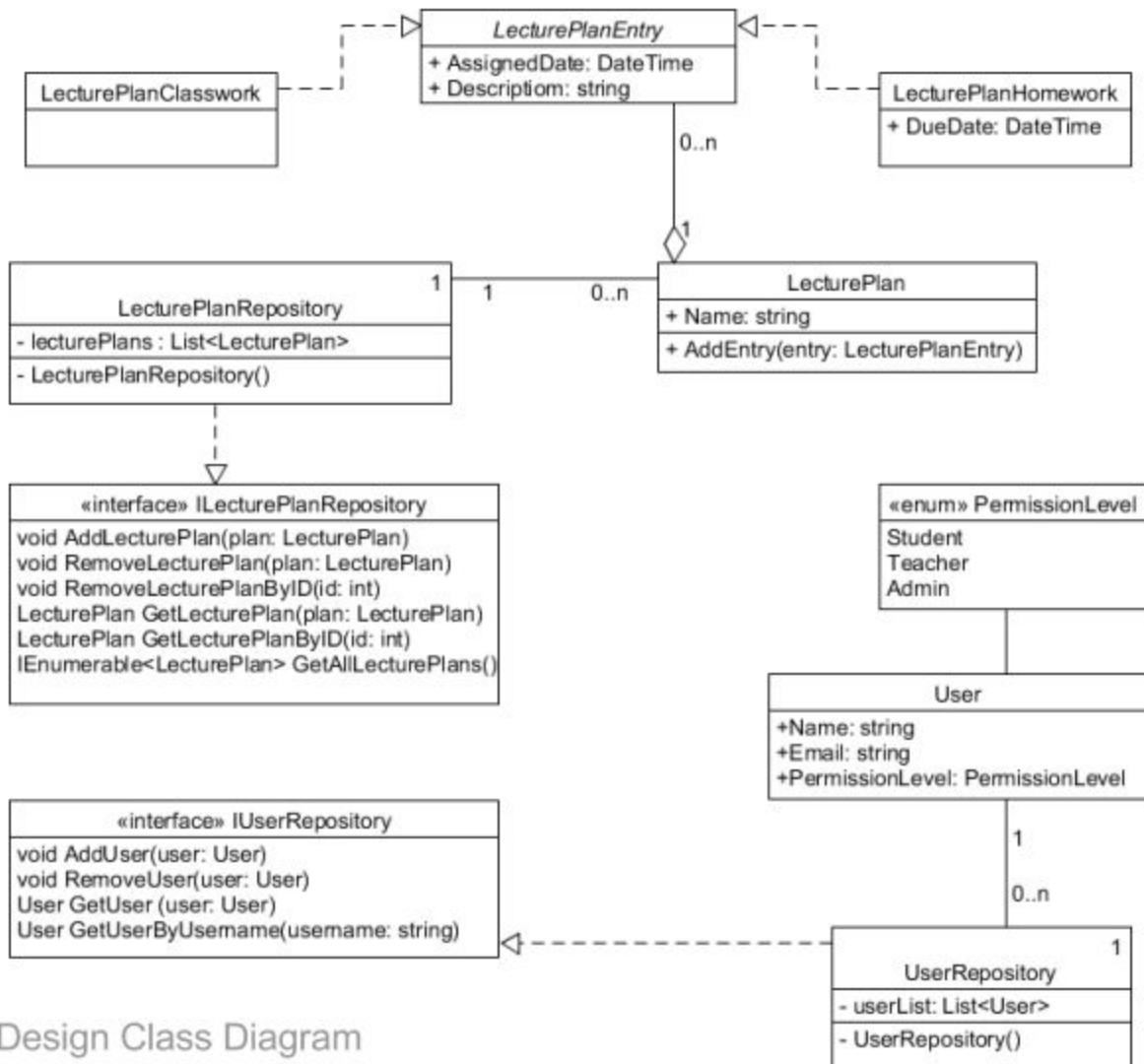
In order to better visualize the relationships between our classes, objects, properties and methods, we decided to create a Design Class Diagram.

Despite the fact that Extreme Programming does not support this kind of form of documentation, we considered it to be a useful artifact for us to get a better understanding of how our classes are interacting with each other and also to get an overview on how we are going to build our software. The quality of our DCD is assured by a set of criteria we have created for this artifact. During the quality assurance step, the DCD will be checked against these criteria, seen in Appendix section 11.6.

An alternative to the Design Class Diagram would have been the Domain Model, which shows the conceptual classes and the interactions between them. Despite this, we decided to choose the Design Class Diagram over Domain Model because the DCD contains the conceptual classes plus software classes as well, hence this artifact contains more useful information we can use. The reason of not creating both of the artifacts is because our system turned out to be not very complex, therefore it is not difficult to visualize what the real classes in our system are.

In Extreme Programming, creating a Design Class Diagram goes against the idea of XP, being a rapid methodology, it slows down the development process and makes it more difficult to drop a feature, as now multiple diagrams have to be edited. Despite this fact, our purpose is also to create high-quality documentation which will also support traceability and will be a plus to our learning process.

During our study time we have worked a lot with this artifact, but for this project we decided to change it and create a new way of working with it. A example could be, the usage of the "void" type, which Larman says should not be included into the artifact. We decided to change this mainly because we would like to get rid of the confusion of not having anything for the void type. We decided that by doing this it would be more helpful for us to better understand the diagram. More information on our discussion about this, see Appendix section 11.6.



Design Class Diagram

Development

We internally deployed our Web API to Azure as tasks were completed. You can see our reason for choosing Azure in the section: *Deployment - Choosing hosting solution*. This was mainly since having the Web API helped the development of the phone app significantly.

This had the added bonus of discovering many of our bugs during development, when we tried to interface with the Xamarin project. Sometimes we did have to rework things in the phone app, due to changes in the Web API.

We sometimes hit bumps in working on the app, due to missing features in the Web API, therefore, in the future we wish to finish off the Web API, before moving onto the phone app, as the Web API is the more central piece.

Pair Programming

After the documentation is finished and we have enough tasks to know what to do, it is time to start coding. Pair programming is one of the practices of Extreme Programming and for the programming part of this project, we decided to adopt this practice. We are going to be working two people per workstation, switching frequently, so all of us can be familiar with the whole system and gain more experience.

Hardware

The entire solution was developed on the personal computers of the members of the development team. The app was tested and run directly on multiple Samsung Galaxy devices. This was to ensure that the product would work on the devices of the users. If, for whatever reason, a physical device could not be used for the testing, the Android SDK Manager was used.

Development Environment

For the development of this project we used the Visual Studio IDE. We chose this IDE because it is the one that we are most comfortable using and we have been using it throughout our studies. It also has many good features and tools that make development of a project easier. We were able to use Xamarin tools in Visual Studio for the development of the app.

Source Code Control

To help ensure proper backup and logging of our activity, we decided to use Git for version control and GitHub for remote storage.

GitHub is a very popular tool among developers and is widely supported, which is why we decided to use it.

Some other version control systems we could have used are TortoiseSVN, TortoiseSVN is an Apache Subversion (SVN) client, implemented as a Windows shell extension; TortoiseCVS, TortoiseCVS lets you work with files under CVS version control directly from Windows Explorer; or Mercurial, Mercurial is a free, distributed source control management tool.

SVN vs GIT³

The main difference between GIT and SVN is that GIT is a decentralized system, while SVN is centralized. SVN has a control server which has the main repository, and everyone checks out from and commits to this repository. GIT is decentralized, so every member has a full copy of the entire repository.

If a GIT repository is lost, only the unique changes that had not been committed are lost. In SVN, if the central server fails, the entire repository is lost. The advantage of this centralised approach is that if someone is kicked off the team, it is easy to lock them out of the repository.

As everyone has a full copy in git, it also faster to commit, change branch, and merge, as it does not need to communicate over the network.

SVN also has a tendency to take up alot of space. Every commit is stored in a file/folder structure. For example, Mozilla's firefox repository is over 12 GB today, over the 240.000 commits made the past 10 years or so. However, you can just checkout individual folders, so if not everything is relevant for you, simply download the required folders. With Git, you always have to check out the entire project.

No Source Control

Another alternative would be to simply have no source control system at all. Developers who follow this approach have developed some techniques which can imitate a SVN repository.

Some of the ways people work without source control is to do regular backup. When we investigated the web development company WEB2IT, this was the approach they followed.

Some software, such as Unity game engine, does not play well with Source Control solutions. When we had to work with Unity during a school event like game jam, we shared a folder across the network, and worked of that, similar to SVN. Alternatively, we could have passed around a USB.

When only one person needs to work on the software through its lifetime, using a source control system can seem like unnecessary overhead.

Testing

Testing Tools

Unit Testing (Microsoft Framework)

³ <https://www.codeforest.net/git-vs-svn>

We chose to use microsoft's built in framework for unit tests, as it comes with visual studio and suits our needs. An alternative to this could have been to use the framework by NUnit, but this requires additional setup. For our purposes, either would suit our needs, and the differences between them are minimal.

Test Driven Development⁴

We implemented the test driven development process because we believe it provides a lot of benefits for the quality of the code and the development process and because this is also a practice of Extreme Programming.

When you create unit tests and run them often, it becomes much easier to fix any bugs that are created because, as long as you are testing often, only a few minutes of code will need to be redone.

Having tests allows us to change parts of the code without fear of breaking something, because with units tests, we will immediately know what broke and where. This allows much easier refactoring and code improvement.

Testing also acts as a kind of documentation. If anyone wants to see how to create a specific object or connect to a specific API, they simply need to look through the tests because it will be shown there.

TDD encourages decoupled code, else you cannot properly create Unit Tests. If the code does not have dependency injection, you cannot unit test. This is good practice when writing code and TDD can make this easier.

An alternative to TDD is to focus on the integration or system testing. This will save time and can lead to some better code design because often, when TDD is being used, the development team only wants to get the green bar and will do anything to get it even if it is not done the best way. TDD also causes programmers to create many useless tests because if the test can not fail, other than the first time, there was no information gained from the test, so it is basically useless.

This approach, however, can lead to more errors and bugs in the code because you are not able to test each specific function of the code. It can also make debugging more difficult because

⁴ This site was used for some of the information in this section:
<http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

you can not look at the specific test and determine exactly what failed, you can see the general area where something went wrong, but you still have to find it yourself.

User Acceptance Tests

Acceptance tests are one of the most important type of testing in XP. Acceptance tests are used to ensure that the requirements have been met and the system is acceptable.

The way XP does this is: Every sprint acceptance tests are created and the user stories are being translated into these tests. The tests represent an expected result from the system. Acceptance tests are also used as regression tests before a production release. Customers are responsible for verifying the correctness of the acceptance tests and prioritize which of the failed tests are of the highest priority.

We are aware of this type of test, but we did not implement them in our project as we did not have enough ready functionality for the P.O. to actually test, hence the acceptance tests were not needed in our case. In a real life scenario, we would use this type of testing to make sure our product owner is satisfied with the software by testing it himself throughout the project and also to have a say, making sure everything is correctly implemented.

Deployment

We developed this software with the goal of being a prototype version, for them to experiment on how they may wish to do this for future versions. We will be handing them our source code and artifacts in a zip, together with a copy of our report as it has some advice, such as our arguments for various hosting solutions.

If we were to have deployed this, we would likely need to train the teachers currently using the google site, to use the new system.

We would have to find a way to deliver the app to the students. With Androids we could simply distribute the APK file freely, however with iphone, we are required to have an apple developer account to publish the app through.

Our software would be replacing portions of their current system, so we would have to figure out a reasonable cutover converting the current documents.

Our suggestions would be to provide some guides and tutorials on how the new system should be used, both from an administrative perspective, but maybe also for the students. Hopefully, the User Experience and User Interface would be good enough for this to be redundant.

Choosing Hosting Platform

The hosting platform of our choice during the development was Microsoft Azure. What this meant for our development phases, we talked about under the *Development* section of this report. Here, we will focus more on the advantages and disadvantages of Azure and the alternatives.

The main reason behind us picking azure, is that we as students have free tiers of their hosting service, gained through Microsoft Imagine. Azure is also well integrated into Visual Studio, making deployment a one-click action, and since Azure is a cloud based platform, should demand require, scaling up would not be an issue.

However, we do not have physical access to the servers, or even VPSs the application runs on. We have to place our faith and trust in Microsoft that they do not partake in sketchy actions with our data, meaning we cannot 100% guarantee to live up to some of the EU data protection laws. By not having access to the servers directly, there is very little we can do in order to protect against various attacks, like man in the middle. If someone takes over the network connection at Azure, we stand defenceless. It is also difficult to assess the security of the SSL certificates.

Yet, we feel and believe that for our purposes Microsoft's services will be secure enough, and that they are credible enough to handle their servers' security. We do not think that using their services is an issue, nor will be in the future.

Alternatively we could have gone with Google's Cloud services, facing virtually the same pros and cons, as we still have to trust them, plus we would have to pay, as their service is currently business only, without students programs.

If we had gone with traditional hosting, there could have come issues in regards to scaling in the future, as you pay for a fixed spec server. This means you either need to pay more money to handle peak loads, whose resources then get "wasted" during less heavy periods. With traditional hosting, you usually still do not have physical access to the servers, other than on a software basis.

Many people often go with home-, or self- hosting, having dedicated servers at the office. This allows you to easily monitor the servers, in order to detect attacks, and you can more easily guarantee the process in which data gets treated. Having a self-hosted system requires you to hire administrators to setup and maintain the servers. Servers can also be quite expensive for high quality setups and can drain a lot of electricity. Self-hosting still hits issues in regards to efficient scaling as talked about earlier.

Maintenance

We developed this application as a prototype, hence we have given little thought toward maintenance. During development we did implement the SOLID principles, so, if our prototype software expand into the full system, maintainability should be less of an issue.

If our system were to be the full system, other than the generic tasks, such as adding new features, server updates, etc., our system would require various archival tasks. Since lecture plan entries are simply piled on top of each other, the database may become quite big over time.

Our recommendation would be for LærDansk to develop a new system. A common phenomenon, The Second System Effect, as talked about in “Beginning Software Engineering⁵”, how the first version usually ends up being canned, since you learn about a lot of things you wish to do different. They could use our prototype to see how they could actually do it.

We are handing over all our documents, research and source code, and in case they have questions, our contact information.

For maintenance tasks, we could have included a feature in our software to submit bug reports the users experience. This would be our main source for corrective maintenance. For perfective maintenance, we could have drafted agreement with LærDansk for more hours, otherwise our code should be solid enough for others to take over. We would like to be able to offer preventative maintenance as well, mostly in order to fix potential security exploits before they become issues.

The User interface, or app, would have to be updated through the respective app stores. Both Android and iOS has auto-update features built in for these.

⁵ Stephens, Rod. *Beginning Software Engineering*. John Wiley & Sons, 2015.

In regards to our Web API, we would simply have to push the new code into the production environment. Ideally, this should have gone through staging before, for us to test the new iterations of the software, before publishing.

Retrospective & Conclusion

For our final retrospective, we will take a look at our overall project. As part of our PDCA, we look at what we did this project, and what we can carry over to our next project.

The first major point we came across is that Extreme Programming ended up not being the best methodology for us to try and follow. We had expected that LærDansk would be much more inconsistent about which features they wanted and in what priority order, but it only took one meeting to formalize the requirements, and then another meeting to confirm them again. After the initial meeting, we managed to get some decent wireframes drawn, and after that there were no requirement changes.

We now realize that a more iterative waterfall SDM would have been more appropriate, as the requirements did not change. Redoing this project in an iterative waterfall would allow us to more eagerly define our High Level Design and plan better for the User Interface and we would be sure that no further features would be randomly added or removed, and we would likely have been able to deliver a more working program, but with fewer features.

Problems & Issues

One of the more time consuming administrative tasks we spent quite some time on were the issues on github. To increase our traceability in regards to who has done what, we used the issues from github as our tasks and a github project for a kanban-style board.

This also meant, when submitting a commit, we could tag the issue and they would be automatically linked, however, this was not always done. We often had to go back over issues and the commit history to find when certain things were added and manually write a comment on the issue to refer to the commit.

This has lead to a slight decrease in developer traceability as we may have missed commits and going over the commit history, it is not automatically clear which issues the commits are attached to.

This is the first time we have tried this approach, hence it was expected to not go entirely as planned, yet it was an annoyance nonetheless. For future projects, we believe writing down stricter requirements to commits may help deal with this issue.

Compared to the expectations of Extreme Programming, we have drawn way too many diagrams. We went further into detail on our Design Class Diagram than necessary and drawing many System Sequence Diagrams as well. We did this in order to increase our traceability between requirements and the code-base, practice drawing the diagrams ourselves, and because we did not feel confident enough in our skills to simply keep all this information in our heads, yet.

We skipped out on a lot of the actual programming, mostly due to us feeling under time pressure and wanting to finish up more for the System Development report. We did however manage to learn and research a lot in preparation for our coding, such as learning about the Xamarin framework for phone app development. We also felt, as our programming is part of an independent exam from this project, our priority for this should be the system development aspects. The main consequence of this is that we do not have anything near a working demo project to actually hand over to LærDansk, but we do have plenty of material for them.

Our log keeping abilities were also subpar, and very few logs were kept. We believe this is because our project work is mostly done when we are together, hence we simply talk about any issues, changes or decisions. Sometimes we try to write it down, either in an issue or alike afterwards, but this means a lot of information can be lost.

We have discussed how we can improve this in the future. Our initial idea was to strictly enforce rules about discussions having to be formalized as an internal meeting, but we feel that this may inhibit our creativity and in-the-moment ideas. An alternative could be to continuously record our conversations, so we can listen back over for important details, however issues of privacy were raised for this. At this point, we have not concluded how to combat this issue.

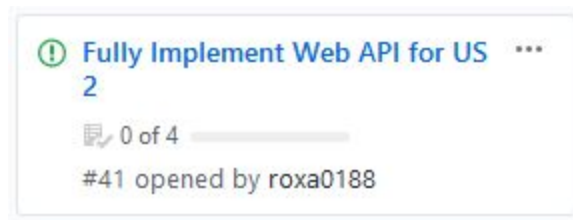
What went well

During this project, we spent much time researching the criteria for the various artifacts we were producing. We created checklists in the github issues, which were filled out as quality assurance was being done. This is where most of the logging problems also occurred. While having many discussions about specific criteria, the reasons behind changing them were not always properly recorded.

We also feel confident about our initial setup to using Extreme Programming. We felt prepared to start this project using XP..

Getting project rooms at school has been a major challenge this project, with rooms being booked well into december, and hallway space being loud. We decided to start working at Roxana & Jonas' place, as this was quiet and much more comfortable. We even feel like our work efficiency went up after not working at school anymore.

Despite our problems with using the github issues tagging, our general feeling from using github projects and issues is quite positive. Tasks being automatically moved to QA when closed and into the tasks if failed and reopen. While we still have a lot to learn about GitHub, we feel more confident using the software.



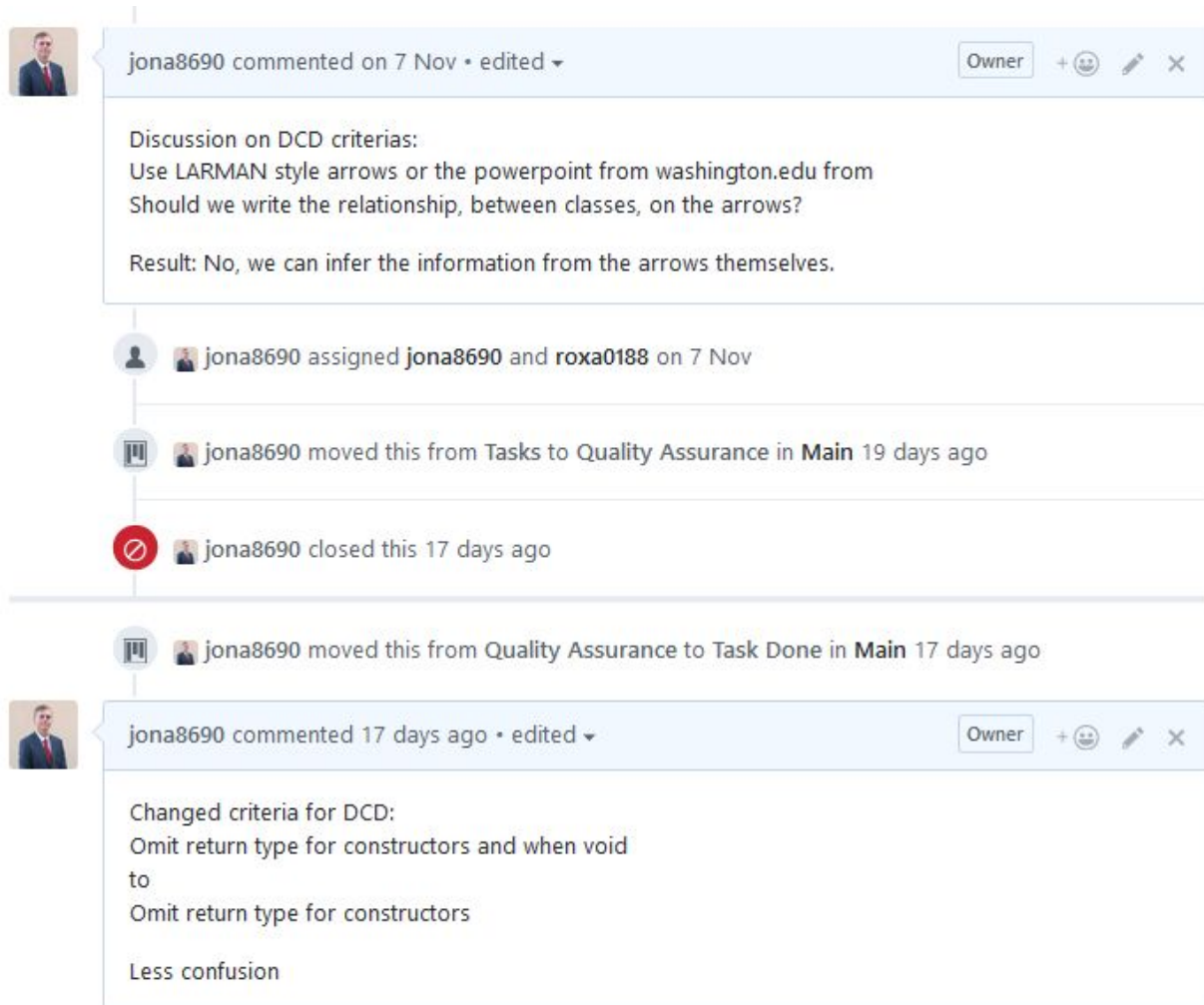
In this project, we also did much better in creating tasks than we did in previous ones, with almost all work having had a task created at some point. Tasks felt sufficiently small to be managed without being overly simple as “create this line of code here”.

The main advantage of collecting our decision logs and tasks in github, was that it all ties together. When doing Quality Assurance, we could copy-paste in a checklist, which would then show up as a progress bar on the task in the kanban board, as seen in this example.

The screenshot displays the history of a Jira issue. The actions are as follows:

- jona8690** moved this from **Tasks** to **Quality Assurance** in **Main** 21 days ago.
- jona8690** assigned **hedv0149** and unassigned **Matt2694** 21 days ago.
- jona8690** commented 21 days ago • edited. The comment text is "Was closed with 190144c". The comment is owned by **jona8690**.
- jona8690** closed this 21 days ago.
- jona8690** commented 10 days ago • edited by **Matt2694**. The comment text is:
QA:
Admin & Teacher Select Class
 - ☒ No colours (distracting)
 - ☒ No visual elements (distracting)
 - ☒ Title
 - ☒ "How to get here" description
 - ☒ Concept/Purpose/Objective of this frame
 - ☐ Content & Data to be shown in the frame
 - ☒ If/Then behavior (description), what happens if
 - ☒ Navigation Links / Buttons
 - ☒ Error ConditionsThe comment is owned by **jona8690**.

Above is how an issue would evolve over time. Using issues is great for tying a developer to their actions, and to keep logs over them. There is a larger example in Appendix section 6. We also kept decision logs to our issues, making it very easy to backtrack on the reasons for the changes, e.g. when we changed our requirements for the DCD, so that if we ever wondered why our specifications are different, we could easily track back and see why. Example seen below.



The screenshot displays the history of a Jira ticket. It begins with a comment from user jona8690 on 7 Nov, discussing DCD criteria and LARMAN style arrows. This is followed by an assignment of the task to jona8690 and roxa0188. The ticket was then moved from 'Tasks' to 'Quality Assurance' in the 'Main' column 19 days ago. It was closed 17 days ago, indicated by a red circle with a slash. After 17 days, it was moved back from 'Quality Assurance' to 'Task Done' in the 'Main' column. A second comment from jona8690, dated 17 days ago, details changes to the DCD criteria: omitting return types for constructors and voids to reduce confusion. The comment box includes an 'Owner' label and icons for adding comments, editing, and deleting.

jona8690 commented on 7 Nov • edited ▾

Discussion on DCD criterias:
Use LARMAN style arrows or the powerpoint from washington.edu from
Should we write the relationship, between classes, on the arrows?

Result: No, we can infer the information from the arrows themselves.

jona8690 assigned jona8690 and roxa0188 on 7 Nov

jona8690 moved this from Tasks to Quality Assurance in Main 19 days ago

jona8690 closed this 17 days ago

jona8690 moved this from Quality Assurance to Task Done in Main 17 days ago

jona8690 commented 17 days ago • edited ▾

Changed criteria for DCD:
Omit return type for constructors and when void
to
Omit return type for constructors
Less confusion

Working without sprints

One of the problems we had with working without the sprints was that we mainly forgot to add deadlines to our tasks, which may be the reasoning behind the lack of code implementation. Other than this, we did not feel any noticeable slowdowns in our workflow.

In the future, instead of putting deadlines on the tasks themselves, putting it on either the Epic or Ready stories to better help guide us through and prioritize our work better.

We did enjoy working with this task based manner. For our next project, we would like to experiment more with this, where we include some more elements of scrum trying to organise us better.