

University College of Northern Denmark

IT-programme

AP Degree in Computer Science

Dmaj0919

SOLVR.ONLINE

SYSTEM DEVELOPMENT

Repository:

Maros Cuninka, Sebastian Holesz, Samuel Horacek, Martin Hotka, Ioan-Sebastian Voinea

Aalborg, 21. 12. 2020



University College of Northern Denmark

IT-programme

AP Degree in Computer Science

**Class:** Dmaj0919

**Title:** SOLVR.ONLINE – SYSTEM DEVELOPMENT

**Abstract:**

**Project participants:**

Maros Cuninka

Sebastian Holesz

Samuel Horacek

Martin Hotka

Ioan-Sebastian Voinea

**Supervisor**:

Nadeem Iftikhar

**Submission date:** 21. 12. 2020

Table of Contents

[**INTRODUCTION** 4](#_Toc59139580)

[**1.** **Product Vision** 4](#_Toc59139581)

[**2.** **Business model canvas** 4](#_Toc59139582)

[**3.** **Personas** 5](#_Toc59139583)

[**4.** **Mock-ups** 6](#_Toc59139584)

[**5.** **Methodology** 7](#_Toc59139585)

[**5.1.** **Theoretical comparision of plan-driven and agile development** 7](#_Toc59139586)

[**5.2.** **Our approach** 8](#_Toc59139587)

[**5.3.** **How we used XP** 9](#_Toc59139588)

[**5.4.** **How we used SCRUM** 13](#_Toc59139589)

[**5.5.** **Our conclusion** 17](#_Toc59139590)

[**6.** **Planning & Estimation** 18](#_Toc59139591)

[**7.** **Risk analysis** 25](#_Toc59139592)

[**8.** **Requirements definition and Quality Assurance** 27](#_Toc59139593)

[**8.1.** **Functional** 27](#_Toc59139594)

[**8.2.** **Non-functional** 27](#_Toc59139595)

[**9.** **Configuration management** 29](#_Toc59139596)

[**9.1.** **Version control** 30](#_Toc59139597)

[**9.2.** **Change management** 31](#_Toc59139598)

[**9.3.** **Release management** 31](#_Toc59139599)

[**CONCLUSION** 31](#_Toc59139600)

[**BIBLIOGRAPHY** 31](#_Toc59139601)

[**APPENDIX** 32](#_Toc59139602)

# **INTRODUCTION**

Text…

# **Product Vision**

When we, as a group, were faced with a decision to come up with a unique idea that could change an industry while being suitable for our project, we came with a range of ideas. The technical requirements laid out by the curriculum had to be considered, as well as the applicability of the solution to the real world. Thus, we allowed ourselves some time to innovate and we narrowed the ideas to just a top 3 from which we voted the winner. Most of the ideas were related to ecommerce stores or customer-to-customer (C2C) marketplaces with which we decided to go. None of us had ever worked on a C2C project before, so we took it as a challenge.

The product vision is something that shapes the project itself – it describes the overall mission of the product. For us, it meant that each member had something to fall back on, something that forms the end-goal of the entire process. In the end, we came with the following statement:

**“The ultimate marketplace where your homework finds its Solver”.**

It has to be noted that before coming up with our vision, we focused on the product's name. We felt that it needs to be striking and that its domain should be available for us to buy, which we did straight-away. After some considerations, we ended up with Solvr.Online.

# **Business model canvas**

After we had come up with the idea for our product we decided to find out if it is even possible and would make sense for us to start with it. For this reason, but also a better understanding of the entire idea, we created a business model canvas which helped us to visualize the core business concepts.

As you can see in Appendix A – Business Model Canvas, on the right side we have the internal factors we considered as relevant for us – Key Partners, Activities, and Resources. In the middle, we have a Value Proposition, which describes what values we want to deliver, and on the right side, there are external factors, such as Customers and the Market in general.

The main goal of the business model is to plan how a business intends to make money and the business model canvas makes it easier for stakeholders to understand it. As our project was not meant to be developed in such a way, we have not considered this to a greater extent than what was described before.

# **Personas**

In our case, there are more ways how the customer could interact with our product. A user of the product is either a Poster who needs the assignment to be solved or a Solver who is going through posted assignments and looking for the ones he can solve.

According to this, we created personas which are fictional characterizations of users that represent real groups of potential customers. Creating these Personas helped us to better understand how the user would interact with our product and what group of customers we are targeting. We would describe them by name, photo, age, education, job, interests/hobbies.

Referring suddenly to a Persona with a Name instead of just a generic user (Poster/Solver), gave us a clearer vision of who we are developing the product for. It helped us to keep realistic ideas of users. We would commonly say that “Muhammad (Figure 1) would use the product this way because… “. Although Jan (Figure 2) would use the product in another way. We would also think about the motivation behind the personas using the product. Either a busy elementary school student who does not have enough time to solve all of his homework or a high school teacher who wants to earn money on the side by solving assignments.

Text

Description automatically generated

Figure 1 Persona

Text

Description automatically generated

Figure 2 Persona 2

# **Mock-ups**

The first rule of UI that we applied when creating the mock-ups is the principle of the common region. It defines most of our screens, especially when the user is creating or solving an assignment. The two pages mentioned above contain forms that are separated into regions, each region with its own defined scope. The separation of those regions makes the form clearer to the user and what he/she must fill in.

The second principle that guided our mockups is the principle of similarity and simplicity of target. Our business model is based on commission, when a Poster posts an assignment, we deduct a fee from the gross value. Therefore, the easiest action on the platform should be posting an assignment. The “Post Assignment” and “Publish” buttons are separated by position, color, and shape from everything else on the page, making them different and easy to find. Besides that, the “Post Assignment” button is placed directly on the navigation bar, being always available to the user.

On both pages, Create an Assignment and Display an Assignment, the Deadline, Credits, Academic Level, and Subject have similar functionality. To make it easier for the user to understand that we have combined all of those in a separate region called Details.

You can see our mock-ups in Appendix B – Mock-ups.

# **Methodology**

## **Theoretical comparision of plan-driven and agile development**

In order to develop a system or a product, one has to look at the way the entire development process will look like. How will he ensure that the system will not have to be remade in the middle of the development process and ultimately how will he meet the customer’s deadlines & expectations?

There are two main approaches: a more traditional, plan-driven development & more dynamic, agile development. The choice often depends on the internal circumstances (such as budget, team size, time to market, security, and reliability of the product, which can be crucial for a government system, etc.).

Plan-driven development tries to eliminate all the uncertainty and risks as early as possible. Overall, the process is very predictive, as well as thoroughly documented, meaning that it is quite formal and does not leave much space to change requirements. The process is what matters the most. This development method is based around development stages, with each having a certain output at its end. Three of the most prominent plan-driven methodologies are the Personal Software Process (PSP), Team Software Process (TSP), and Rational Unified Process (RUP).add reference

On the other hand, agile development is, as the name suggests, much more about agility and dynamicity. It tries not to predict as much, but rather adapt to change because it assumes, we cannot get things right upfront. In contrast to plan-driven, it is all about the value, not so much the process. The formality is also much lower, meaning the number of ceremonies is lowered to a minimum. What matters is teamwork, strong collaboration with the customer, and flexibility.

Agile has many frameworks that are used all over the world. The ones that we will focus on are SCRUM, Kanban & Extreme Programming (XP). SCRUM puts more focus on its artifacts and ceremonies. The development team uses artifacts such as the product backlog, with user stories to be implemented or the sprint backlog with user stories for a specific sprint (iteration). SCRUM also has three formal roles (SCRUM master, product owner, development team), whereas Kanban has none. Kanban also omits most of the SCRUM ceremonies and focuses purely on visualizing work on its Kanban board.

XP is the only purely software development framework. It is described by its four values (communication, simplicity, courage, and feedback) and twelve principles. All of them, as well as their implementation throughout our project, will be discussed in greater detail later in this report. In short, it focuses on frequent releases, programming features only needed at each moment, coding in pairs, and constant refactoring.

## **Our approach**

During this project, we tried out several agile methodologies, practices, implemented many values from XP or SCRUM, but the core of our processes was guided by what is called the “Agile manifesto”. We have followed the 4 points of the bespoken manifesto every day, from sprint 0 to the final hours of our project. To show you how we implemented them in our project and everyday developer life, we will present them one by one.

**Individuals and interactions over processes and tools**

Our team is all about communication. We stay in touch at all times, we report to each other our steps, progress, and decisions. Like this, we are able to keep the team working perfectly on both individual and group levels. We have stepped away from long documentation, complex diagrams, and redundant protocols and rather embraced conversation and teamwork.

**Working software over comprehensive documentation**

We realize that each team has a time budget, which is all they get to develop a piece of software. In our case, we had 5 weeks, which we decided to spend on developing working and comprehensible code, rather than spending many hours on writing documentation and thus losing much of the limited time budget

**Customer collaboration over contract negotiation**

This point may have not directly affected our project, or at least it may seem like that, but if we take a closer look, we can see that throughout the whole process we prioritized communication and discussion about features and functionality rather than needless bureaucracy. We had to be our own customers, but we feel we acted accordingly.

**Responding to change over following a plan**

Change. Change was the subtitle of this project. To count all the changes would be a laborious task and to say the least, we changed the whole architecture of our project 2 times, which meant redoing and refactoring hundreds of lines of codes and tens of extra working hours spent on it. But in the end, the courage to embrace change helped us to develop a product we were all satisfied with.

## **How we used XP**

During sprint 1, we took advantage of the methodology of (XP), its practices, values, and procedures. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels [1]. That means long sessions of code review, pair programming, etc.

**Our approach to XP**

XP concentrates around 5 values and 12 practices [1]. We have tried all of them but focused on only those that worked for us or applied to our project. We pushed those that worked for us to the extreme as advised by the creators of the XP methodology. Let's go through them one by one and see how we implemented or embraced them.

**The 5 values of XP**

1. **Communication**

In our team, which has been together for longer than a year now, we value communication above everything else. We are not only teammates, we are also friends, housemates. We are in touch every single day and we like spending time together. It only feels natural to us to voice our concerns, talk about obstacles and problems, and help each other with solving them. It is not uncommon that our Teams calls take many hours (the longest being almost 14 hours). Even with the pandemic hitting hard we have managed to use virtual channels to meet and work together. We always try to discuss our steps, work as a unit and just frankly, speak.

1. **Simplicity**

This may have been a struggle for us in the beginning. A simple design is an easy thing for advanced programmers and software developers because it is easy for them to distinguish what is and what is not going to be needed. We have, nevertheless, tried to write our code as simple as possible, do not code “for the future” and only hunt for the functionality needed at the moment. In the end, we have all agreed that we have made enormous progress in this regard, spending the final weeks on coding only the “right here, right now” functionality.

1. **Respect**

Respect, as understood by XP, is not only internal (within the team) but also external, as the developers should respect the client’s decisions and opinions, managers respect the developers and that they are the ones developing the system, etc. In our case, the respect was only internal, but on a high level nevertheless. We know what everybody is capable of. Each of us respects the other team members for what they bring to the table. Those are the fruits of our long-term cooperation.

1. **Feedback**

In the case of feedback, we received both internal (on developer-to-developer level) and external after every sprint on the sprint review meeting, when we were asked questions not only by our teachers and supervisors but also other teams. We have always presented the whole picture and as many details as possible as soon as possible. We have felt like the earlier we can get complex feedback, the more agile we can be.

1. **Courage**

Courage is an important part of all agile methodologies. It is essential that the teams are not afraid of changing the direction of where the project is going. We think that our actions and decisions during the project demonstrate how courageous we were. It is not an easy decision to change everything in your code, leave no stone unturned. And we did so twice.

**The 12 principles of XP**

The following 12 values are ordered by how much we have embraced them, with the ones that we embraced the most being on the top of the list.

1. **Refactoring**

Refactoring was second nature to us. We have refactored from the smallest bits of code such as local variable names (from id to userId to be more understandable) to complex conditional logic, for example, the logic of who can access certain API endpoints. From everybody, through everybody who is logged in, then everybody who is logged in and is a customer all the way to only the person who is logged in, is a customer and posted that specific assignment. The process of refactoring has been embraced at all times and was generally considered the most important one.

1. **Pair Programming**

This is a practice we followed even in previous projects. We know the advantages, but also the disadvantages of it. During the first two sprints, we have programmed in 2 groups, one consisting of 2 and one of 3 members. This was very helpful in the early stages of the project when we all needed to set our minds on how we want the code structure to look and how we want the code standards to be implemented. In the later stages, not every coding session was done in pairs, but at least 75% of them were.

1. **Collective Ownership**

Collective ownership means that every developer can change whatever they desire to, even though it was not them writing the original code. We have embraced this practice more and more as the project was progressing. In the end, all developers were refactoring and changing code if they spotted a mistake not thinking if that specific line of code was “theirs”.

1. **37-Hour week (regular working schedule)**

We have tried to keep the working schedule as regular as possible. Every day at 9:30 AM, but sometimes at 9 AM sharp, we met for a daily standup, where we discussed the necessary matters, later proceeding to the coding itself. In the beginning, we have agreed on 30 working hours per week, but the number of hours was raising with every sprint, reaching just a little above 200 for the last one. The burnout did not come as we all saw the end of the project approaching.

1. **Coding Standards**

We have followed the general C# coding standards [2] when writing our code, if we slipped at any time, we always made sure that we refactor the code later. This way we kept the code clear to everybody.

1. **Metaphor**

Metaphor is applying words, labels, tags, or stories to various elements or chunks within the programming process [3]. We have applied such words to our system to make it easier for us to talk about it. For example, the user, who is a customer and creates an assignment, becomes a Poster, whereas the user who is a customer and solves that assignment, becomes a Solver. This is only applicable with regards to that specific assignment and the roles can be therefore easily switched when talking about a different assignment.

1. **Simple Design**

As mentioned before, when talking about the value of Simplicity, not coding ahead was a bit of a struggle for us in the beginning, but we have managed to reduce this not-agile practice by the end of the project. We have also decided not to focus on the diagrams too much, and only use them in the inception stage of the project, when we were deciding on how we are going to approach it. We also strived for no code duplication, by putting the repeating code into separate methods.

1. **Test-Driven Development (TDD)**

During sprint 1, we have tried out TDD as it is both one of the most praised and notorious practices presented by XP. In our case, we used it during the stage, when the foundations of our system were laid, and we could see its advantage of being certain at all times that your code is tested and also simple, as the TDD helps you to write simple code. Nevertheless, the tests we have written became obsolete as soon as we changed the architecture for the first time. We have decided that TDD created too much overhead and was too hard for us to keep up with it.

1. **Continuous Integration**

When it came to continuous integration, we have tried to follow it as much as it was possible in our small, artificial environment. We did not have any releases planned except for the hand in in the second half of December, and we also did not have any production or testing servers. We at least did the following things:

* Branched our project in a way that main is only for tested and ready-to-show code, whereas the development (and branches that were branched from development) are meant for the implementation of new features,
* We did not have a “clean computer” just for daily builds and integration tests, but we at least implemented gitignore file to avoid the issue of different packages and frameworks installed on different computers, which are usually the cause for the most common explanation of programming issues: “but it works on my computer”,
* Since our project was not so big in its size, we were able to clean and rebuild often, always seeing if there are no build errors,
* The feedback cycle (Develop test case 🡪 code 🡪 integrate 🡪 test) consisted of only the last 3 steps, because we did not develop a test case for every piece of code, we were implementing.

1. **On-Site Customer**

We did not have an on-site customer. In our case it was all about the team, and we were therefore getting feedback about the implemented functionality only from each other.

1. **Planning Game**

The main goal of the planning game is to exchange the information between the customer and the development team. Customer has information about the value and developers about the cost. This was different in our case, because we did not have on-site customer as was mentioned before. Thus, the development team provided all the information about the value of the project with the cost calculation as well. Planning game consists of 3 phases.

In the **Exploration phase**, we wrote down short user stories (more about them explained later), then we prioritized them, and the last step was to estimate. If we could not estimate one user story properly, was it too difficult for us to estimate, we split this epic story into two separate ones. For example, in user story Log in, we had a task Log in using third-party APIs. Also because of time constraints and the difficulties to estimate, we decided to create a distinct user story.

In the **Commitment phase**, customer chooses the content of the first release. We would consider release planning, but we would need more time, somewhere around sprint 6-7. For us, the main focus was to have working code after each sprint with hand-in deadline. This phase was little bit different from what XP dictates in our case.

Last phase is the **Steering phase**. The idea of this phase is to have possibility to change accordingly to customer needs (to our need for us). We chose what to do in the following iteration and divided selected user stories into tasks. Few times we ended up with remaining task which we had to put into the next iteration.

1. **Small Releases**

We did not do any small releases. During the whole project, we were developing what one could call an MVP (minimal value product). It would not make sense to make any releases in the meantime, as the system would not work properly in the production.

## **How we used SCRUM**

Scrum consists of three main roles, Product Owner, Scrum Master and The Team. In our case we as a project group were covering all three roles.

**Product owner**

Since this was a school project, we naturally did not have a product owner. To fill this void in the arguably most important position in the whole SCRUM methodology we all as a group stepped up and came up with ideas in which direction should the product be heading. We all went through the idea generation process, estimation, and prioritization therefore we all felt like this was our project. We managed to replicate his usual duties such as creating the product backlog, setting prioritization of user stories. But some of his responsibilities were not applicable since we owned the product backlog, and the release dates were set.

**Scrum Master**

Coming into the first sprint our idea was to switch the Scrum Master position for every respective sprint. After finishing the first sprint we felt that the position of a Scrum Master was redundant in our workflow, so from that point it was more about who stepped up in the middle of a specific task. During the Unified Process that we tackled with last semester we did not have one assigned Team Leader who would call the shots, there would be someone occasionally stepping up to the leader role, but majority of the decision were made democratically, this was the way that we already had experience with for over a year and worked best for us, so it came in naturally.

Looking at it retrospectively we found ourselves in situations where having the Scrum Master position and utilizing all his powers would definitely help us. One of the mentioned situations would definitely be management of the task board during the sprints.

During two sprints we ended up with tasks that were not finished so usually in a situation like this the Scrum Master would intervene and manage the tasks accordingly.

Another example would be that some discussions took significantly longer because we were missing that one person who would pull the hand brake and say “enough, we do it this way”.

**Team**

Coming into this project we were comfortable with our team composition. Our team has been working together for over a year, so we knew our strengths and weaknesses. Some members excel at frontend design others at backend programming, so based on that we could assign roles in the project. But we also like to learn and grow in different fields in which the pair programming practice helped us considerably. Our biggest strength as a team is that we trust each other, and we are confident in our individual decisions. We are self-organized so we always decided how and when we would work.

With our development methodology based on SCRUM we naturally included the 4 ceremonies which can be also called meetings in our development process.

**Sprint Planning**

Our usual sprint planning started every Tuesday around 9AM. We would open the Product Backlog, look at the User Stories with the highest priorities and discuss the possible combinations and outcomes. After choosing the highest priority user story we wanted to keep it cohesive so the question always was what other user stories would make a good combination with it. We would also look at our velocity and every time our aim was to target values around 30 velocity. After choosing all the user stories for the sprint backlog we would look at them again as a whole and discuss any doubts. In the next phase we would take the individual user stories one by one and create Task for them accordingly. If we felt a task was too complex we would divided it into smaller tasks, until preferred atomicity. When assigning value of difficulty to tasks, we would often choose one task which was in the “middle” and based on that information we would assign effort values to other tasks. We used the Fibonacci number scale for representing the numbers of difficulty. At the end when everything was set and prepared we would start the sprint. We would commonly refer to the sprints by the name of the most prioritized user story, for example “WPF Sprint” or “Authentication/Authorization Sprint”.

**Sprint Review**

In the sprint reviews we would always present our latest work from the previous sprint. The reviews were very beneficiary to our group project, it helped us to get the much-needed feedback from the teachers. Besides the teachers we got also asked questioned by our fellow classmates. We were also on the other side of the table where we would ask questions. For example, we were unsure of the row version implementation but we discussed it with one of the groups and later with teachers. That gave us an idea and assured us that we were on the right track. Getting all this feedback made us realize some possible improvements and helped us to develop an overall better product

**Daily Scrums**

Daily scrums were a huge part in our development process. In our case we could call it a hybrid between daily scrum and daily stand-up because we used some practices from XP as well. We would meet almost every day around 9AM. The main questions were: “What did I do yesterday?”, “What will I do today?” and “Do I see any impediment that prevents me or the development team from meeting the sprint goal?”. We would take random turns until every group member would answer these questions. After that we would inspect the sprint task table and use these three questions as a guide to see how we are progressing so far and if anything needs to be adjusted. We would also move the tasks that were finished from the previous day from “In progress” to “Done”. If everything was to our satisfaction, we would end the daily scrum by assigning each other by preference into groups for pair programming. When the groups were formed, we would start working on developing the product and repeat the same process again in the next day.

**Sprint Retrospective**

Every week when preparing for the sprint review, we would do the sprint retrospective. While preparing the presentation for the upcoming day we would ask ourselves “What went wrong this sprint?” and “How could we improve it for the upcoming sprint”. One of the early issues that we encountered was that we were not fully utilizing the sprint task board, that meant there was confusion in the team on who was working on what. It resulted in two different groups working on the same task, thus creating a conflict in the version control and losing valuable time. We solved this issue in the sprint retrospective by agreeing that when someone works on a specific task, it is properly labeled “in progress” in the sprint task board.

During our sprints we heavily relied on the three artifacts from the Scrum methodology.

**Product backlog**

Arguably the most important scrum artifact was created at the start of the project. It is typically maintained by the product owner but in our case, we were the product owner. This meant that we were responsible for the maintenance and prioritization of the user stories. After creating all the user stories, we could think of, we would add them to the Product backlog. The next step was to prioritize them. We were assigning them a number between 0 and 99 based on the importance of the user story. We did not add much user stories throughout the project because we included everything important at the start.

**Sprint Backlog**

At every sprint planning we would create the sprint backlog based on the contents of the product backlog. It would consist of user stories with the highest value. Apart from sprint planning we would always return to the sprint backlog during daily scrums to track our progress.

**Burnup Chart**

Burnup chart is a graph that helped us to analyze the progress of the work in relation to the total story points for the current sprint. We were tracking tasks rather than points which in our case told us more about what other developers are coding and how well we are doing as a team. Sometimes if we had just looked at our burnup chart during the sprint, we would have noticed that we had no chance of finishing all tasks. But because we did not do so we were left with unfinished tasks at the end of the sprint. This means that we did not utilize the burnup chart to its full potential.

In the figure XXX you can see our burnup chart for sprint 4. The yellow line shows total number of story points, for this sprint it was 35. The blue line shows average or the ideal progress which should be made. The green line shows our actual progress. As you can see, at the beginning of the sprint we were behind the schedule, which was caused by changing the overall architecture. We were dealing with this user story for 2 days, but after we were finished with it, we were constantly above average, therefore we added one more user story to this sprint – Change profile picture (changed the overall velocity by 1).

Chart, line chart

Description automatically generated

Figure 3 Burnup Chart - Sprint 4

**Sprint task table**

This extra artifact was very important for us. We were using it daily. If a group member would take a task, he would mark it “in progress”, this way we could easily track who is working on what tasks, to avoid two separate groups working on the same one. When tasks were finished, we would mark them as “Done” during the daily scrums.

## **Our conclusion**

Looking back, we started off by following purely XP practices. It took us few days to get accustomed to their “extremeness”. After sprint 1, we chose to follow Scrum, whilst borrowing some of the attributes from XP, we thought will help our development process. In the end, the formal structure of the project was shaped by Scrum, but the coding itself by the coding principles of XP.

From our experience, if we are to compare unified process used in previous semesters compared to frameworks used in the project, the biggest difference could be felt at the start of the project. We were not spending considerable amounts of time on planning. We have assessed potential risks and worked up some crucial artifacts, but most of our time was spent on the actual development. Difference was also felt on daily meetings, called “daily stand-up” in Scrum terms, which were much briefer than previously. We discussed purely issues at hand and tried to minimize attempts at predicting future.

# **Planning & Estimation**

Each of our iterations (or sprints if we are talking in SCRUM terms) consisted of couple of common steps and processes. Below, you can see those steps in the figure XXX.

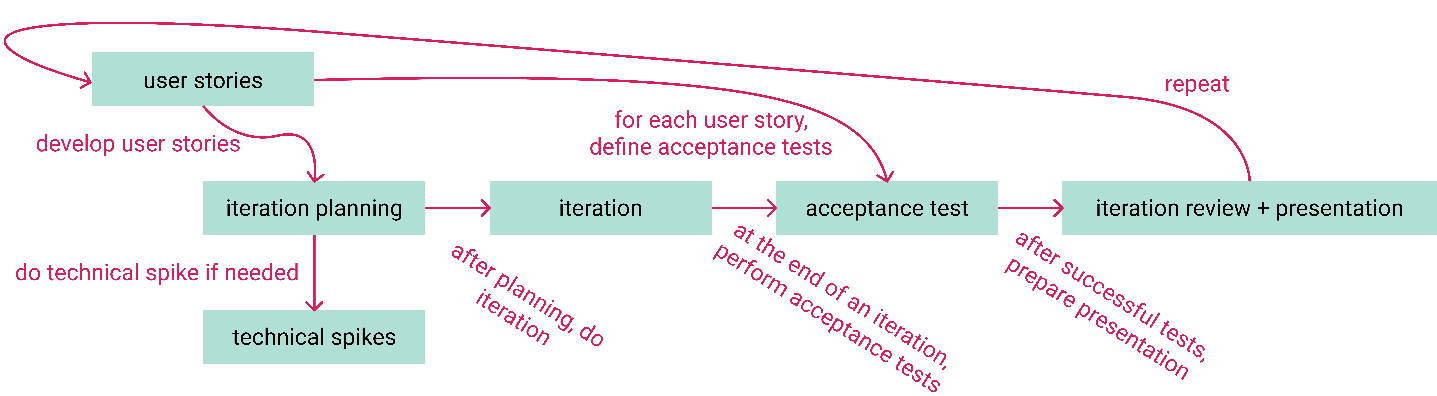


Figure 4 Iteration timeline

1. Developing user stories

The process of creating user stories is described in a greater detail later in this report. We have mainly focused on the 3 C’s and the INVEST values.

1. Iteration planning

We planned every iteration, also estimating how many user stories (in story points) are we able to develop during the next iteration. You will also be able to read about our estimation later in the report. We have mainly focused on the estimation techniques of analogy, and breakdown and calculation.

1. (Optional) technical spikes

Sometimes, there was a need for a spike, especially when working with technologies with which we had no previous experience. Those spikes, which result we usually threw away, were a helpful tool what brought us closer to understanding some more complex technologies, such as Asp.Net Identity, JWT or others.

1. Iteration

During this, in time the longest, step of all, we spent several days on developing the defined user stories.

1. Acceptance test

After each iteration, we have performed the acceptance tests we have created together with our user stories at the beginning of the iteration. Since there was no customer, who usually both defines and performs them, we had to do this ourselves. Successful acceptance tests meant fully implemented user stories and a green light to proceed to the next step.

1. Iteration review + presentation

When the acceptance test was concluded, on every Tuesday we had a presentation and iteration (sprint) review, where we showed our progress, what we have developed, how did we improve our development process and answered questions to the outside world, in our case, the supervisors and other groups. This was a great check for us, as it had an enormous influence on many of our decisions.

**Our User Stories**

A user story is a tool used in Agile software development to capture a description of a software feature from an end-user perspective. A user story describes the type of user, what they want and why. A user story helps to create a simplified description of a requirement. [5]

1. Using business language

A user story should be understandable to the customer and should use business language rather than technical words. We have focused on keeping the user stories easy to understand, even though we had no customer who would write them or who could be there to tell us, if he does understand them. Below you can see an image XXX of one of our user stories. As you can see, we used business words such as Poster, Solver, solutions, assignment, where Poster and Solver can be considered a part of our metaphor as explained before.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 5 User story - Post an assignment

1. The difference between User stories and Technical stories

There are times when developers need to write down some technical requirements or improvements that should be done to the system in the sprint being planned. To avoid using technical jargon in User stories and confusion what is meant as a next feature (User story) and a technical enhancement in the background, not entirely visible to the end-user, we also introduced another type of story provided by agile methodologies: the Technical story. As it was already mentioned, they are meant for the developers to write down the planned technical upgrade. Below you can see an example of one of our technical user stories.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 6 Technical user story - Set up version control

1. The difference between User story cards and Use case descriptions

Last 2 semesters we have worked with plan-driven and UP, where we used Use case descriptions. There are several types of Use case descriptions, the fully dressed (being the most elaborated one), the casual, and the brief use case description (being the least elaborated one). The difference between User story cards and Use case description is the complexity of them and the amount of written description. Whereas the Use case description (namely the fully dressed ones) describes all “happy”, secondary, and error scenarios and do not leave anything for the conversation, the user story only writes down the condensed description in a couple of words and leaves everything for conversation and negotiation.

1. The three C’s of User stories

The three C’s of User stories: card, conversation, and confirmation were proposed by Ron Jeffries in 2001 to distinguish “social” user stories from “documentary” requirements practices such as use cases [6]. We will present both the definition and our implementation one by one

**Card**

The Card is a written description of the story used for planning and estimation [7]. In our case, we used a tool called iceScrum, which provided us with a nice interface and helpful tools to create user story cards. Our cards (as visible in figure XXX and XXX) consisted of:

* Title
* Effort (our estimation on how long it will take to implement the story estimated in story points)
* Value (how much value does the story have to the project)
* Text definition (how we defined the user story, we used the predefined templates)
  + As a <user> I want to <do something> in order to <achieve some goal> (User story)
  + In order to <some goal> we need to <do something> (Technical story)

**Conversation**

The goal is to build a shared understanding of what the feature is. By discussing it with the product owner/customer, developers are able to define the user story. Since there is not much of written documentation for user stories and unlike the Use Case definitions, there is only the happy scenario, developers need to ask questions that help them better understand the feature being implemented. In our group, we like to discuss everything thoroughly so every user story we put down was well thought through and we all had a great understanding of what the feature will be when implemented.

**Confirmation**

The confirmation is the process of developing a set of acceptance tests for each user story. Those are put together by the product owner, as well as they are performed by him when the user story is fully implemented to test if it was implemented in the correct way. In our case, there was no customer, but there were acceptance tests we first wrote down and then carried out as you will be able to read later in this report.

1. The INVEST rules

The acronym INVEST helps to remember a widely accepted set of criteria, or checklist, to assess the quality of a user story. If the story fails to meet one of these criteria, the team may want to reword it, or even consider a rewrite (which often translates into physically tearing up the old story card and writing a new one) [7]. We will now present examples of how we followed these rules.

**Independent**

User stories should be independent, meaning it should be possible to release the features represented by them without depending on other stories. We aimed to achieve this by splitting them by CRUD functionality (Post an assignment, Delete an assignment, etc.), users (Post an assignment, Post a solution) and different logical parts of one complex functionality (Log in, Login using 3rd party login). Nevertheless, the stories were still dependent on each other in the meaning that it was not possible to solve an assignment if there were no assignments, etc.

**Negotiable**

As mentioned before there is not much of written documentation for a user story and it is mostly about the conversation and negotiation. This we followed to the maximum.

**Valuable**

Every user story we wrote down was valuable and if there was a real customer, they would all have business value for him. If there was something that needed to be implemented but did not have an obvious business value for a potential customer, we kept it as a different type of story, such as a technical story

**Estimatable**

We had a rule that every user story which estimate would be too high (above 21 story points) and therefore hard enough for us to estimate would be split into smaller parts. This way we kept all the stories estimatable.

**Small**

As mentioned in the previous points, no story was estimated at 21 story points or higher. We kept them small at almost all time, but there were times when even though the story was not too big to estimate, we noticed later, that it could have been split into more atomic stories. For example, the previously mentioned example of “Log in” and “Log in using 3rd party login”. At first, they were kept as one story, but they were later split into 2.

**Testable**

As one of the three C’s dictate, one should always come up with a clear confirmation (acceptance test) for every user story. We kept our stories testable by defining a set of tests that would confirm the feature was implemented. Let us talk about that part of the process now.

When we had finished with defining almost all user stories, we had to prioritize them, in order to start with the most important ones. During this process we followed H. Kniberg [4], who suggested using value (from 0 to 99 in our case) instead of priority, where the priority number one is the most crucial. By that, we gave our two core user stories the highest value 99 and 90 and other user stories we evaluated from 0 by tens (10, 20, 30...). When we have added a new user story, we could then easily place it in the middle of two different values, so we did not have user stories with the same value.

Before each sprint, on a sprint planning meeting, we picked the user stories with the highest value and the process of estimation started. We thought about tasks it will include and then we used the Fibonacci numbers to estimate them (Figure XXX).

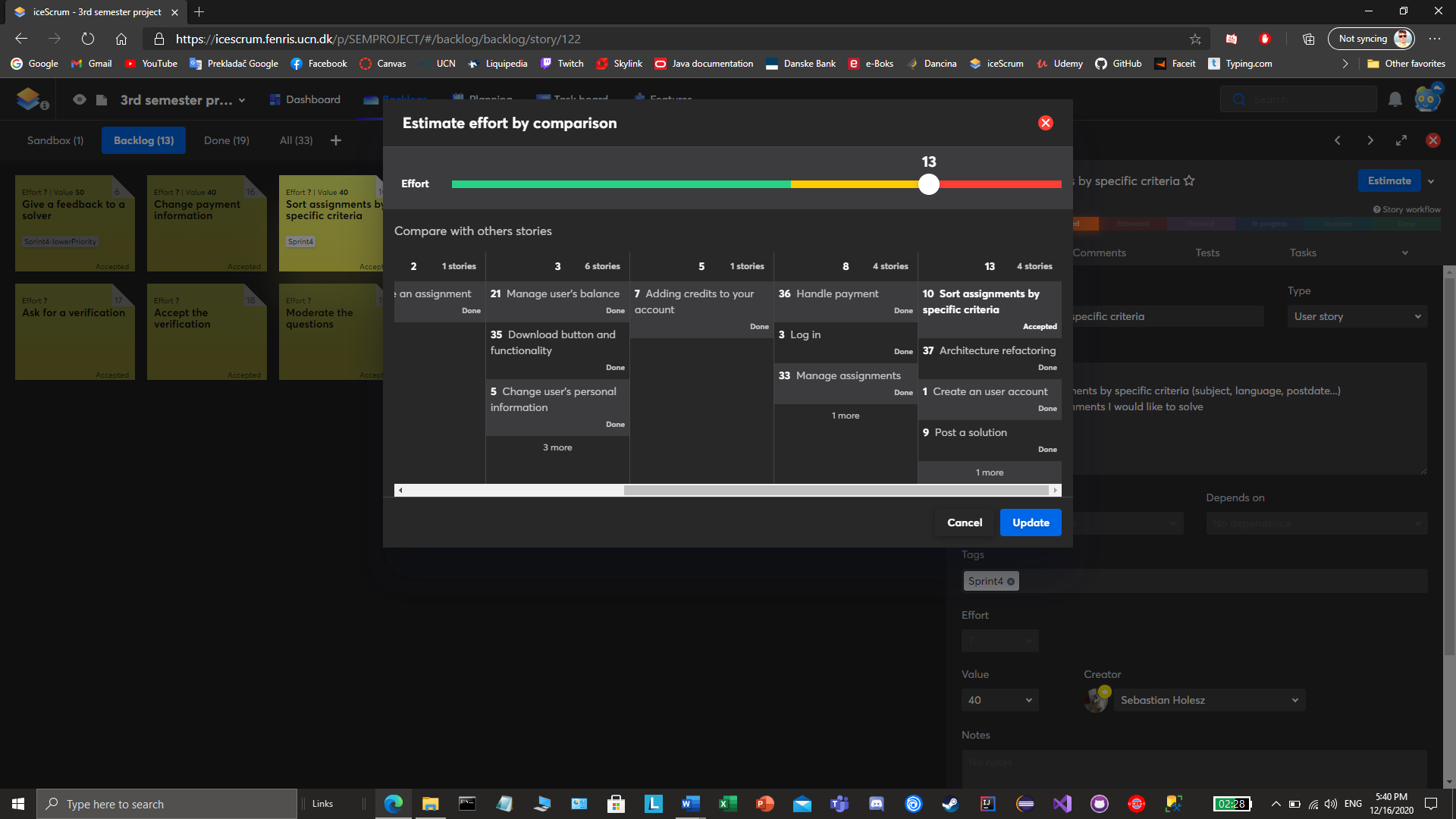


Figure 7 Estimation process

We used a breakdown and calculation method when we ended up with a user story with value higher than 21, we split it into two different stories. It was very difficult for us to estimate the first sprint, therefore we based our estimation on the best guess which was mainly influenced by our previous projects. After that we ended up with a sprint velocity of 26. On the sprint retrospective we agreed that we finished everything but did not have time to implement a new user story. On the other hand, we wanted to challenge ourselves, thus for next sprints we used an analogy planning method and planned the entire sprint with very similar, but slightly higher velocity (sprint 2 - 29, sprint 3 - 32, sprint 4 - 35). As you can see, we increased the velocity by nine between sprint 1 and sprint 4, which was caused by changing the overall architecture in sprint 4. Unfortunately, we did not play planning poker, because as we mentioned above, we got rid of epics and during estimation we all had similar opinions. Also with same knowledge, when we knew almost nothing about topics we were about to face, the numbers would be almost the same.

Next step during sprint planning was to divide whole user story into small tasks, which were then grouped up into 3 categories – TO-DO, In progress and Done, so we could divide them among us and work separately without conflicts. XXX shows tasks for the Post an assignment user story.

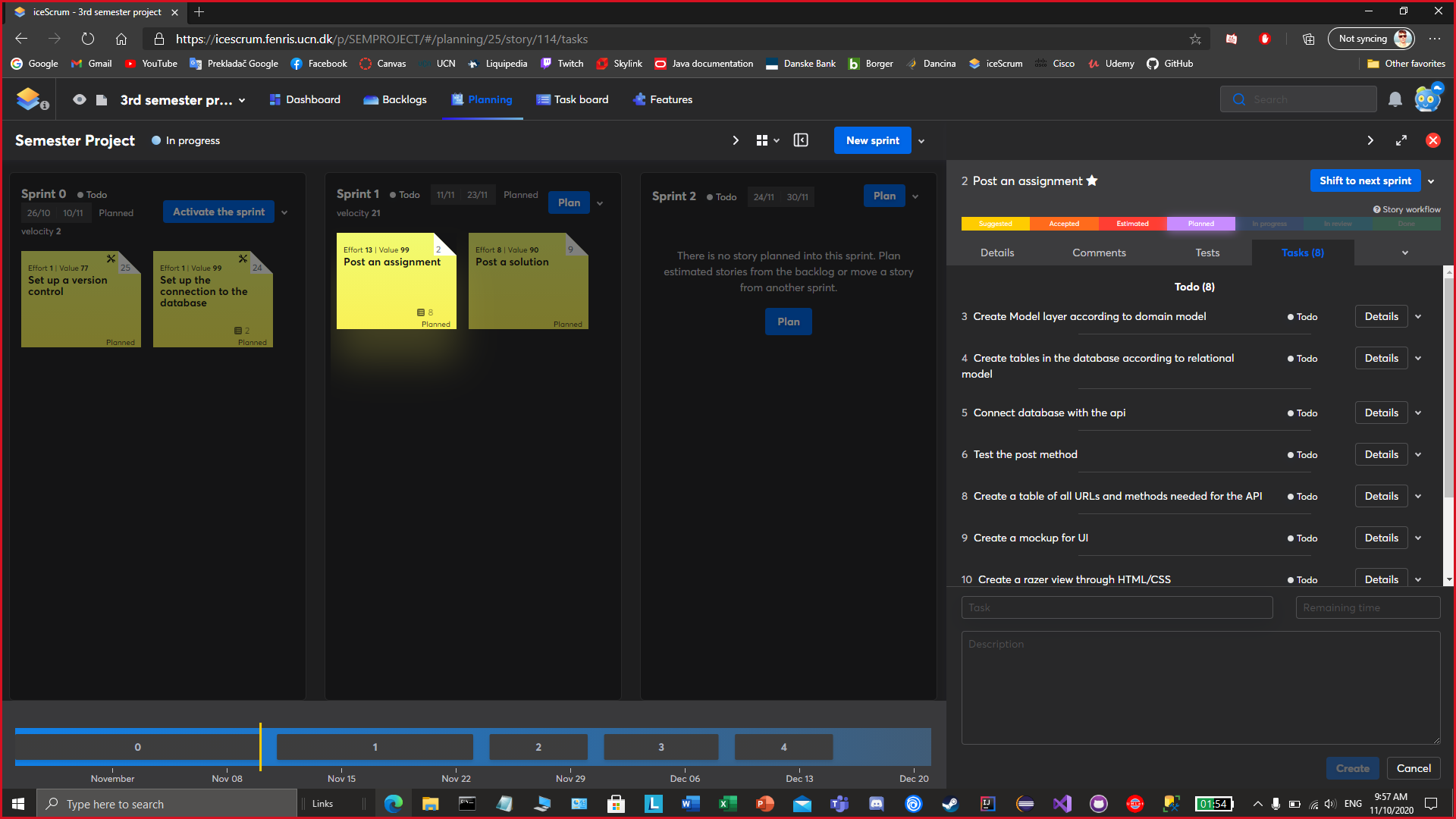


Figure 8 User Story - Post an assignment

During each sprint and on sprint retrospective we used Burnup Chart. It is a graph that helped us to analyze the progress of the work in relation to the total story points for the current sprint. In the figure XXX you can see our burnup chart for sprint 4. The yellow line shows total number of story points, for this sprint it was 35. The blue line shows average or the ideal progress which should be made. The green line shows our actual progress. As you can see, at the beginning of the sprint we were behind the schedule, which was caused by changing the overall architecture. We were dealing with this user story for 2 days, but after we were finished with it, we were constantly above average, therefore we added one more user story to this sprint – Change profile picture (changed the overall velocity by 1).

At the end of the project, we had also in mind what would be the next features we would like to implement. For next releases we would add:

* payments and other third-party APIs,
* forum page,
* a mobile app or make it optimized for mobile users.

# **Risk analysis**

Test Risk analysis can be best described as a process of assessing the likelihood of an unfavorable events that can happen during the project and can influence its success and lead to failure.

We started with risk analysis at the beginning of the project, even before sprint 0, by identifying the risks. You can see the result in the table XXX in first column. After we found out what risks can affect us, we analyze the likelihood and the consequence of each one of them and by multiplying these two values we got the overall score (risk priority).

The top three risks are very connected to each other, because we were using completely new methodology, which is different from plan-driven, and our planning was divided to sprints. From beginning it was very difficult to estimate, therefore the high probability and impact. Even though, there is a corona pandemic, we set the impact to very low, as we already worked in a situation like this in our previous project. We think that this is even a good experience to the future as well, because we could try how is it to work only online, instead to face it our jobs for the first time. Last two risks are brought from last project, where we also encountered burnout at the end of sprint and not all of us are using Windows.

Table 1 Risk analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Risk** | **Probability** | **Impact** | **Score** |
| Wrong total estimates | 8 | 7 | 56 |
| New system architecture | 6 | 9 | 54 |
| Wrong sprint estimates | 8 | 4 | 32 |
| Fluctuant sprint velocity | 8 | 3 | 24 |
| Illnesses | 7 | 2 | 14 |
| Corona pandemic | 10 | 1 | 10 |
| New programming language | 1 | 9 | 9 |
| Problems with new SD framework (methodology) | 2 | 4 | 8 |
| Burnout | 2 | 4 | 8 |
| Different OS | 1 | 5 | 5 |

Next step was to plan how to minimize the effect of these risks as a part of qualitative risk analysis. We did not plan everything ahead, but we were monitoring it continuously during the entire project. We have only written down the strategy for the most crucial risks. For the new system architecture, the strategy was to find out what suits us the best and ask for a feedback. Nevertheless, we planned this, we end up with changing it two times, which had great impact. With the estimations, we decided that in the first sprint we will not overestimate ourselves, based on that we chose the starting sprint velocity and after that we knew if we could deal with higher or not. Monitoring crucial risks is shown in table XXX.

Table 2 Risk monitoring

|  |  |
| --- | --- |
| **Risk** | **Indicators** |
| Estimation | Good estimation at first sprint without changing the velocity by much |
| Architecture | Changed architecture 2 times |
| Illnesses | very low impact other members substituted |
| Different OS | dealt by booting to Windows and by changing .net framework to .net core |

# **Requirements definition and Quality Assurance**

The requirements of a system are the description of the needs that the system must fulfill, but also the constraints that define the product. These requirements reflect the necessities that the customer or/and the user have. A requirement can vary in many ways, from an abstract statement to a more detailed description. It can be written in a natural language as user requirement, so that the system user can understand them, or it can be written for system developers.

## **Functional**

The functional requirements are the WHAT of the system, they often describe what the system should do.

Following an agile approach, we chose not to have a formal document stating all the requirements, as the requirements may change over time and it would get outdated as soon as that happens. Therefore, for this project, our functional requirements are described by the product backlog, containing user stories for each individual requirement. The user stories are speculated by us, based on the personas that we have created prior to this.

As we have chosen our own project, challenges did not emerge during the requirement definition, it was easy for us to completely understand the system and create complete user stories. The language used was one understood by all team members, and the requirements did not change throughout the development process.

## **Non-functional**

The non-functional requirements are the HOW of the system, they describe how the system should respond and interact with the user.

For our project, we have set a list of non-functional requirements that we tried to fulfill. The challenge here was quantifying the result by testing the system. One of the most important non-functional requirements for the user is usability. In order to test that, we have conducted tests with 5 potential users. The test was composed of 4 tasks that each user had to do using our system for the first time:

* Create and publish an assignment.
* Find and solve an assignment.
* Add credits to your account.
* Find the assignment and solution that you posted.

For all the tasks, we have measured three factors. The first one was completion rate, it shows if the user finished the task successfully or not, the result of this was 100%, meaning that all tasks were finished by all users.

The second factor measured was the number of errors, for each task made by the user, any wrong button pressed or any hesitation from the correct workflow is counted as an error. For the first two tasks, the average number of errors was 1, some of the users having trouble with setting the assignment deadline correctly. For the third and fourth tasks, the number of errors was higher, reaching an average of 3 errors. The users were having trouble with finding the input box to insert the number of credits, but also with understanding the ordering system of the assignments in “My Solutions” page, making it hard for them to find their solution.

The last factor was satisfaction level, after each task, we asked the users to rate the workflow based on their satisfaction, the rating was done from 1 to 5. All the ratings we received were between 4 and 5, the smallest one being due to the inconvenience described during the previous paragraph.

Besides the usability requirement, we have come up with a list of non-functional requirements that we have described and tried to set a measurement criteria for them, as shown below.

Table 3 Non-functional requirements

|  |  |
| --- | --- |
| Security | The API layer should separate the FE from the BE making it impossible to access user data directly.  User should be forced to use secure password to lower the chance of account hijacking. |
| Usability | The software solution should be intuitive and easy to use.  The system should not have any “dead ends”. |
| Performance | The system should not make the user wait for response for more than 200ms. |
| Maintainability | The code should be well-arranged and easily modified. |
| Testability | All tiers of the system should be easy to test. |
| Robustness | Robustness against attempts to flaw the system.  Guard against exploits like SQL injection and take care of all input scenarios that could end up in an error/exception. |
| Compatibility | Users should be able to run both clients no matter the OS.  Website should support all top used browsers. |
| Scalability | Modularity of the system so only specific parts of it can be scaled up/out |

## **Quality Assurance**

During the planning process we also considered what qualities we want to follow. Our quality management did not involve any formal documentation [5], instead we all agreed on these practices:

* Code standards,
* Unit tests,
* Acceptance tests (Appendix C) for all user stories,
* Refactoring,
* Naming convention for git branches,
* Not pushing untested code to main branch.

When we felt that we forgot to follow some of them, we went through them again on standup meetings and changed something if necessary.

# **Configuration management**

Systems are continuously changing during development and so did ours. Working on new features, fixing old bugs and overall refactoring were done on daily basis. As these changes are introduced to the software, a new version is created. Without good configuration management it would be very easy to lose track of it.

Configuration management consists of four related activities (Figure XXX) and you can read on how we implemented them in our project [5]:

* Version control,
* System building,
* Change management,
* Release management.

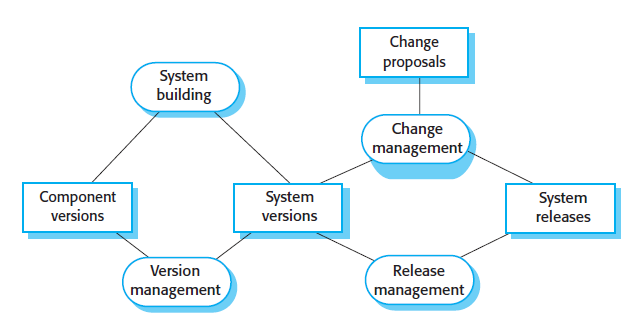


Figure 9 Configuration management activities [5]

## **Version control**

Version control helps the development team to manage their files. It is a powerful resource, because it can track every change in the code and stores a lot of useful metadata about each file - its history, individual changes, by whom were they committed, etc.

Compared to the previous semesters where SVN was our version control choice, in this project we were using git. Compared to SVN, git is not a revision control, it is source code management. Also, contrary to the centralized SVN, git is distributed, so there is not the risk of a single point failure. Git is widely used especially in the open-source community and it has helped our development process tremendously.

One of the main concepts of git is branching. The name "branch" presumably comes from the file tree metaphor and helps the development team to work concurrently and independently on each other. Once each member is done, he "merges" his branch back to origin branches which make the actual source code.

We decided to have the "main" branch, which always had working, production-ready code. In the first few sprints, we only relied on this branch and some feature or bug fix-related branches which were used briefly and later merged to the main. Later, we discussed the option of having another core branch - development, which would carry working, untested code that if stable one day, would be merged to main. We also decided to improve the naming standards of each branch, so just from the name, one could tell what purpose it served. Was it to fix a bug, or to roll out a new feature? It was the norm that feature branch can only be merged to the development one, and we had hoped that bug fix branches would serve the purpose of maintaining the code in main, but it must be noted that we did not deal with this perfectly. Branches including bug-fixing were usually merged back to development to make our workflow a bit smoother. This, however, would not be possible if our code in main was already in hands of the customers who would not have the time for us to roll out a new feature including the bug fix.

In conclusion, using git was a great experience. One can say that its learning curve might be too steep for some, but once learned, git is an essential part of the developer toolbox. What gave us the most headaches were traversing the different branches, merging them, and moving back and forth. Towards the end of the project, we all felt much more comfortable and were able to achieve most of the git related tasks without the help of any internet search engine.

## **Change management**

For large organizations, but also for start-ups, it is very important that evolution of a system is controlled, and proposed changes are prioritized based on cost-effectiveness. [5]

As we mentioned many times in this report, we did not have product owner/on-site customer, therefore all changes came from our own initiative. It would make no sense to write change request form first, therefore all suggested changes were communicated on daily meetings. All suggestions were then voted by each of us and accepted decisions were implemented. During sprints we changed description of some user stories, added a few user stories, whole architecture, .net framework to .net core. These are just few of them.

## **Release management**

The almost last stage of entire process is release stage. *A system release is a version of a software system that is distributed to customers.* [5]We have not gotten this far and for us this system was not even intended to be released to customers. We have focused throughout the entire project on working code, which would be representative at any stage with final “release” on December 21 – deadline.

# **CONCLUSION**

# **BIBLIOGRAPHY**

|  |  |
| --- | --- |
| [1] | D. Wells, "Extreme Programming - Values," 2009. [Online]. Available: http://www.extremeprogramming.org/values.html. |
| [2] | S. Prakash, "Geeks For Geeks," 2020. [Online]. Available: https://www.geeksforgeeks.org/c-sharp-coding-standards/. |
| [3] | B. W., "Explain Agile," 2018. [Online]. Available: https://explainagile.com/agile/xp-extreme-programming/practices/metaphor/. |
| [4] | H. Kniberg, SCRUM AND XP FROM THE TRENCHES How We Do Scrum 2nd Edition, C4Media, 2015. |
| [5] | I. Sommerville, Software Engineering Tenth Edition, Essex: Pearson Education Limited, 2016. |
| [6] | Wikipedia, "Wikipedia - The Free Encyclopedua," [Online]. Available: https://en.wikipedia.org/wiki/Extreme\_programming. |
| [7] | "Agile Alliance," [Online]. Available: https://www.agilealliance.org/glossary/three-cs/. |
| [8] | "Visual Paradigm," [Online]. Available: https://www.visual-paradigm.com/scrum/3c-and-invest-guide. |
| [9] | "SVN vs Git - Javatpoint," [Online]. Available: https://www.javatpoint.com/svn-vs-git. |
| [10] | "What is version control | Atlassian Git Tutorial," [Online]. Available: https://www.atlassian.com/git/tutorials/what-is-version-control. |

https://www.productplan.com/glossary/product-vision/

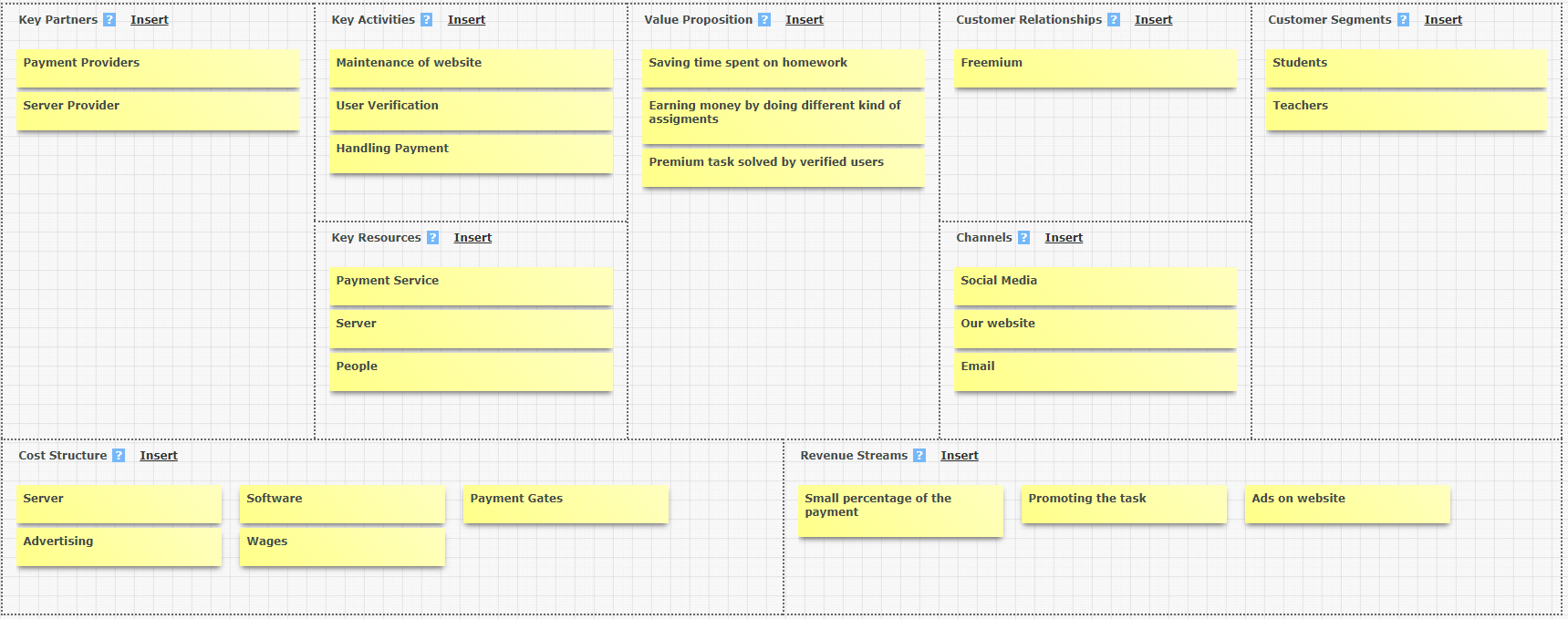
https://www.bmc.com/blogs/scrum-vs-kanban/

https://www.visual-paradigm.com/guide/software-development-process/agile-vs-waterfall/

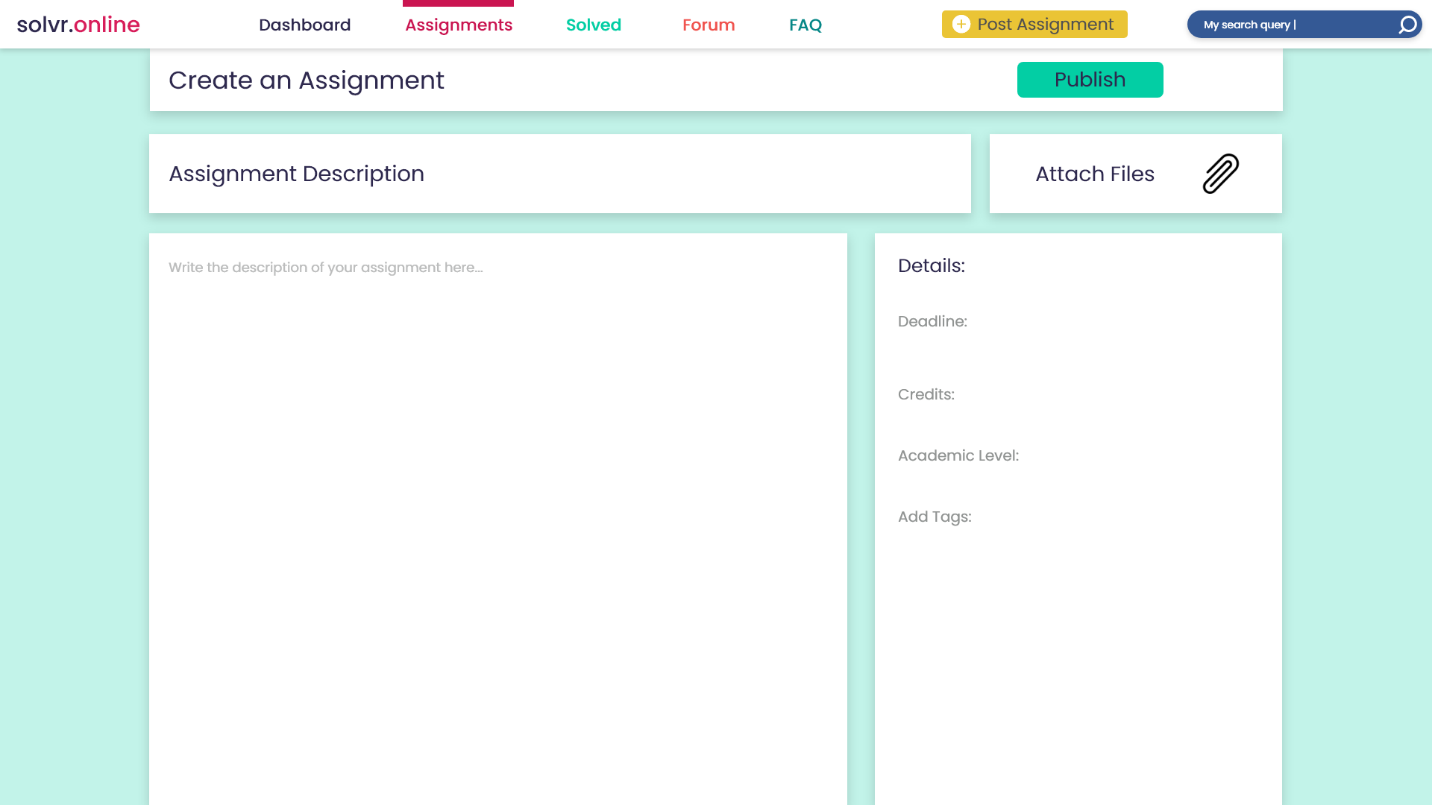
https://en.wikipedia.org/wiki/Extreme\_programming

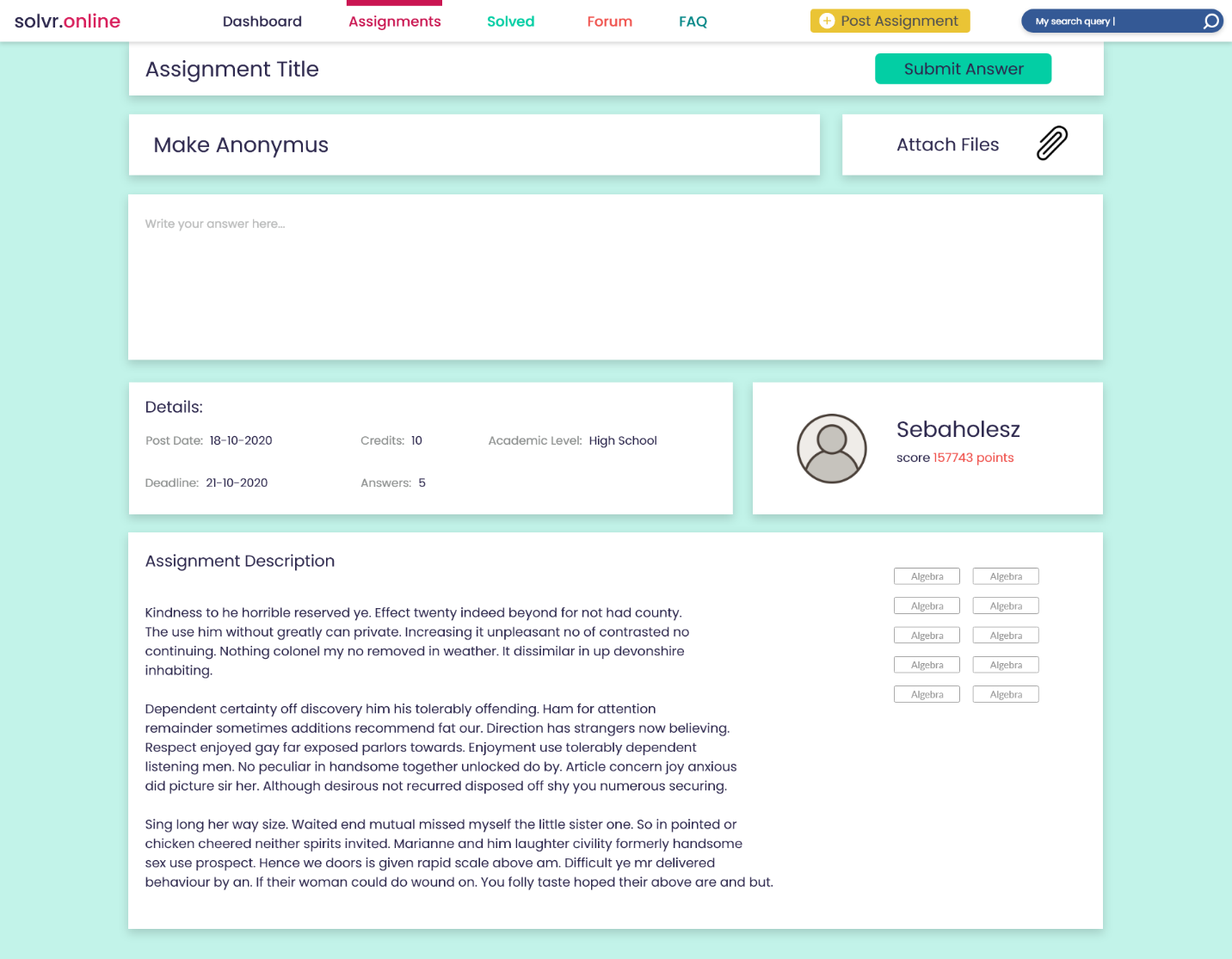
# **APPENDIX**

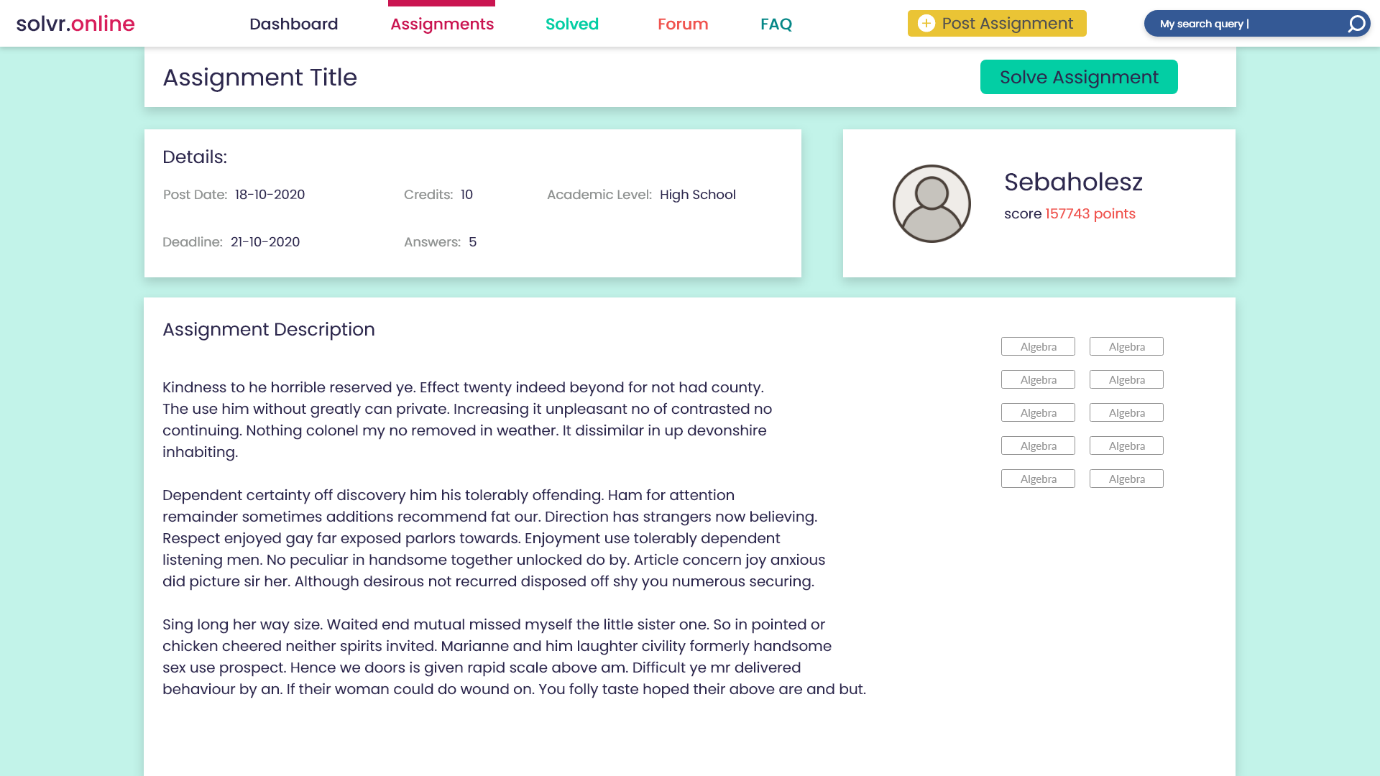
Appendix A – Business Model Canvas



Appendix B – Mock-ups







Appendix C – Acceptance test

Graphical user interface, text

Description automatically generated