Team Reflection - Team Phish

Agile software project management - DAT257 / DIT257

Team members:
Pontus Nellgård, Zakaria Hersi, Jonathan Hedén,
Adrian Håkansson, Lukas Carling, and Johan Fridlund.

Examiner:
Jan-Philipp Steghöfer

Date: 2020-10-30

Customer Value and Scope

The chosen scope of the application under development including the priority of features and for whom you are creating value

The basis of the application was to follow one of the UN's sustainability goals. We chose number 13, Climate action. We felt the application had to be simple yet useful for both everyday- and casual users. We decided to develop a calculator in which you enter some of the more regular CO2 intense activities to both enlighten and inspire the users to better themselves. We began by defining what a complete application should contain and then groom what was not deemed completely necessary for a minimum viable product. Then we further removed features that were deemed too time-consuming or unreasonable.

During the beginning of each sprint, the team met up and discussed the user stories and tasks undertaken in the previous sprint in order to get a grip of what features undertaken were implemented and which were not. The user stories in the backlog were assigned a priority (Low, Medium, or High) and clothing sizes (S-XL) to estimate effort required. Some of the user stories had prerequisites for features (even though they were prioritized as high). This required discussions as to what order the user stories should be assigned. The idea of always providing a minimal viable product (MVP) was always present during these discussions as features that make the user want to use the application was deemed more important for working as an agile team. This was often difficult as the team had to act as not only developers, but also potential users, and the product owner. As is natural, different team members have different ideas and different prioritizations regarding what features are more important and provide more customer value. Therefore, it was very important from the start to make everyone feel that their voice and opinions were being heard and respected.

One example of where creating value was prioritized higher than "writing better code" was made very clear with the supervisor when we discussed restructuring how to handle the database (DB). We wanted to make it follow more principles of object-oriented programming, but restructuring it would not create much more value for potential users. So a compromise was reached where some restructuring was done, but only enough to simplify the developer interface, making it more OOP and more coherent.

In the next project, we will, hopefully, be assigned a product backlog and not have to act as both developers and the product owner. In that case more of the work would probably be dedicated towards translating the product backlogs epics into more graspable user stories. This is because a product owner from outside of the development team might have a more vague description of what features need to be implemented. If we are not assigned a product backlog, maybe there is still a detailed description of what we're expected to develop that could be used to form a product backlog. For size estimation it might be a good idea to investigate other options. In this project a medium task was slightly smaller than two small tasks, and a large slightly smaller than two medium ones. This became somewhat bothersome when estimating velocity which was discussed but not really used. For future projects a

clearer definition of effort estimation would be desirable especially as a way to better determine how the team performs in terms of feature implementation.

Much effort was spent setting up a local DB which made many tasks dependent on it. It works now, but it would probably have gone a lot smoother and faster if some of the features used easier implementation (such as hard-coding) just to test how the features would function and what they would require. To follow patterns such as OOP it would probably become clearer for many developers if there was a class diagram that was continuously maintained.

The idea of first gathering information and defining the overarching scope and functionalities of the application worked out well. This could have been done to a greater extent by writing better epics (if not already assigned from a PO) and user stories early on. By better we mean more understandable from any background, not just a developer one. To become better at estimating actual performance by using velocity the size estimation has to become clearer and more defined as a quantifiable value. This should also make it easier to establish reasonable undertakings for each sprint and in discussions with the product owner.

To deploy features faster we realized that it would be better to work with hard-coding the data required for the features first then once the features behave as desired, move on and implement them in the DB. This will create less conflicts related to the DB as the developer has experience with how the application "should" behave and thereby the requirements posed to the DB. To motivate developers to maintain the structural integrity of the application a class diagram would be a nice complement to maintain pattern development and responsibilities. In order to keep this updated this would have become a regular part of the development process. When new larger features are implemented a meeting should be hosted prior for establishing the class responsibilities.

The success criteria for the team in terms of what you want to achieve within the project
Early on in the course, we all had goals in mind both for the product we were about to develop and for what we wanted to learn in terms of working agile. When it comes to our goal with the product we began, as described above, by shaving off unnecessary or time-consuming features we envisioned in our heads. What we eventually landed on was an application where a user could create an account, login, input their daily activities, and view their carbon emissions. Users could also compare their output to where they should be if they want to do their part to reach the UN environmental goal nr. 13 (Climate action). The last part, to be able to easily view and compare your outputs, was an important part for us. All this together became our goal for the application itself. As mentioned, creating an application was not our only goal with this project. We had all created many applications before in other courses, so what would make this different from all the rest? Working agile in a team using scrum was what would make this different then previous projects and became our foremost goal with the project. In the beginning, it was hard and confusing. We spent a lot of time trying to populate the scum board but were confused about where things should go or how things should be created. We would create tasks that stemmed from nothing and we would assign people to large assignments (user stories), instead of letting everyone pick smaller

tasks freely from the board. The two first weeks were the toughest, but slowly with the help of our supervisor we started to understand how to work in an agile manner and why it was so efficient. Our routine regarding the different scrum ceremonies was, at least in the early half of the course, far from impressive. Sprint planning was the first one we got a good grasp on. Sprint review and sprint retrospective, however, were not properly executed until later in the course.

In a future project, it would be beneficial if working agile had a role as a success criteria in order to establish a more effective and streamlined development process earlier in the project. This goal would become obsolete after having worked agile for long enough but for most of us as of right now it could be very beneficial. It could also be beneficial to set up part-way goals for the scrum process to ensure that progress is being made in the structure of our agile workflow. One part-way goal could be to have an agreed upon structure for the scrum board before a certain period of time into the project. This happens to be something we struggled with and could improve greatly in the future. Having this as a criteria would ensure that a foundation is set early so the team can focus on the project and fine-tune the structure along the way. Furthermore, communication is a key part of agile programming which makes it reasonable to set a success criteria for communicating in different ways.

The list of possible success criterias is long and some are more relevant than others for specific projects, so it can be hard to know which ones to use. To better understand what criterias are applicable and how to use them, gaining experience with different kinds of projects is important. However, even if you do not have any experience, someone else in the team might. Which is why communicating and discussing what success criterias to use within the team and hear as many different viewpoints as possible is always a good idea. When deciding on criterias everyone should not only be informed about these, but also know where to find them later in the project in case they forget. KPIs are of course a great way to gauge progress of different goals. Integrating the most valuable goals into the KPIs would therefore by a great way to both emphasize and measure who well the team is achieving a particular goal. For communication, it would be a great idea to establish many different venues of communication. Daily scrum should of course be an important one but it is also important to use some sort of digital communication platform like Discord, like we had, or Slack.

Your user stories in terms of using a standard pattern, acceptance criteria, task breakdown and effort estimation and how this influenced the way you worked and created value

In the beginning, we had poorly designed epics which stemmed too much from the perspective of a developer, including how to structure the database and various different functions. This was also true for most user stories. They followed the standard “As a <> I want to <> so that <>” format and were assigned priority (Low-High) and size (S-XL).

Each week we reviewed and if needed, adjusted our epics and broke them down into smaller user stories and them in turn to smaller user stories and tasks. After a while, we got the hang of assigning relatively reasonable sizes, but in the beginning, these were primarily focused on

the user stories and not the tasks. This changed as the tasks were made into their own cards. We used the SCRUM board though it appeared rather static as the tasks "lived" inside the user story cards. After some discussions with our supervisor, he advised us to formulate the Epics and user stories in a more readable way for anyone to read and understand. For boosting morale he recommended to break the tasks out into their individual cards making the SCRUM board much more dynamic.

At the beginning of each sprint, there was a lively discussion of which Epics and which user stories would create the most value and be implementable during the sprint. In the middle of the project, we realized we could in a much better way categorize the different user stories and tasks by associating them with a specific Epic. We also made two separate columns in the SCRUM board for sprint backlog-user stories and sprint backlog-tasks in order to get a better overview of which user stories are relevant for the sprint and which tasks are associated with which user stories.

Our way of working with user story priorities could be expanded and linked to customer value in a more explicit way if the end user had been more clearly defined. By receiving feedback from end users throughout the process we could continuously get a better idea of what brings the most value and which user stories to prioritize. These could be included in smaller regular meetings (demos) in which the developers demonstrate the produced MVP of the sprint to the stakeholders and receive feedback. This feedback should then be incorporated into the next sprint thus generating a cycle of customer value improvements.

To better understand how to tie customer value to a specific feature we must first better understand the needs and wants of the customer. By working closer with the customer having regular dialogues this will likely be achieved as the developers get first-hand input on what the customer wants and expects. This could be done by applying extreme programming (XP), which is especially effective when there is a small developer team and there are dynamically changing software requirements. This often requires more from the customer when it comes to developer practices and code knowledge and for them to sacrifice more of their own time.

Your acceptance tests, such as how they were performed, with whom, and which value they provided for you and the other stakeholders

We made a document specifying the definition of done.

- Code is peer-reviewed

This means that the feature worked on is reviewed and free of known-bugs (tested by the developer).

- Code is checked in/put in a pull request

A pull request is made to merge the feature branch into the sprint (developer) branch. The pull-request includes comments about what features have been added and a basic description of how they are implemented.

- Code is tested

The feature branch (pull-request) is tested by an external tester (another developer) who either approves the feature and merges it into the sprint branch or leaves comments of detected issues.

- Code/feature passes regression testing

The features merged into the sprint branch work together with other merged features.

- Code/feature passes smoke testing

Reasonable pitfalls are tested, for example, strict integer inputs can handle users faultily entering a string.

- Code is documented

The features and non-obvious functions are commented. If general solutions that may work for other features are made known to the team members. For example, loading an fxml file (markup language file used in JavaFX) into the centre of the main layout or using a modal pop-up window for “add features”.

When making large merges (at least once a week) all/most of the team is present and observe the merger and together pitch in to solve conflicts. This has worked well, but sometimes there have been schedule clashes. In larger teams this might not have worked as smoothly so a better solution might be for only the tester and the developer to together handle merge conflicts. It would probably be a good idea to have dedicated testers who in two separate sessions review the code. First with fresh eyes without knowing how the code is implemented and only exposed to the API/UI and what the feature should do. Note what works and what creates bugs/crashes. Then another time with the developer to make a comparison of how a new “actual” user might interact with the program and how the developer wants the user to interact with the application. This should be done to make the application more user friendly. Neither of the developers were comfortable writing developer tests (such as JUnit). There exists frameworks for conducting GUI tests in JavaFX and looking into this would greatly improve our capabilities to perform standardised testing.

Dedicated testers will be “expensive”. It is therefore a good idea to keep it in mind when writing the social contract and the definition of done. Ensuring that testing becomes an integrated part of development. This can be a part of simply maintaining properly commented code and clear descriptions of the features when creating a pull request for additional external testers. The team would have to learn how to write and regularly conduct tests to establish

features still perform as intended. This would require time both to learn and also to implement thereby a prioritization has to be made between implementing new features and writing tests. This could be included in the definition of done (DoD) to include, for example, all the most important methods in each module should be tested, or that all public methods that are included in other methods should be tested.

The three KPIs you use for monitoring your progress and how you use them to improve your process

We began by discussing and researching KPI:s online for inspiration. We eventually settled on three which at the time seemed reasonable, but as the project moved along we eventually realized that we didn't use them properly and honestly. In the beginning, we didn't always know how to work with the KPIs or what they were really for. After some great discussions with our supervisor, we settled on three new KPI:s later in the course and started to properly work with them. They are much more relevant and function better with our project. They didn't feel like a chore to be completed every sprint review but instead felt meaningful. Below you can see a table of our old and our new KPIs.

Table 1, key performance indicators

Old KPI:s (not used)	New KPI:s
80% of tasks assigned as Done get Committed.	Individual stress levels on a scale of 1-5 how stressed you feel a) At the beginning of a sprint (sprint planning) b) At the end of the sprint (reflection)
+/- 20% sprint points undertaken are completed each sprint	Individual-level of satisfaction on the overall performance of the sprint. (Satisfaction measured on a scale 1-5)
The number of user stories considered "work-in-progress" (to ensure focus and not spreading the members too thin).	At least 80% of the story points are completed after each sprint. (Small: 1, Medium: 3, Large: 5, XL: 8)

Though not an excuse, the KPI:s were likely not taken as seriously as they should have been early on as there were a lot of new things to learn and try to wrap our heads around. Using KPI:s did not feel like an organic part of developing something, but we now see the benefit much clearer for use in reflecting on different aspects of how we work and understand the team's health to perform better in future projects.

Table 2, Stress levels beginning of the sprint (1-5)

Sprint/ Member	Pontus	Adrian	Jonathan	Lukas	Johan	Zakaria
Sprint 6	2	2	2	1	1	3

Sprint 7	4	5	2	3	2	2
Sprint 8	3	4	4	4	4	1

Most of the team's stress levels increased, for some dramatically, during sprint 7. This was probably an effect of it being the last full sprint before the final demo. A lot of functionality was implemented, but some of the restructurings to the GUI were not merged with that functionality. Some of the functionality at first considered crucial, but later considered not such as recurring events had to either be implemented or "hidden" from the user. This probably caused a lot of stress and led to some overtime, see Table 2. Here a more "realistic" priority had to be taken. While some of the features such as filters were not highly prioritized these were simpler to implement than a complete restructuring of the database which recurring events likely would have required. So even though some features were set a higher priority, given the end of the project coming so soon these were decided not to be implemented (their core functionality is still available though not accessible/visible to the user).

Table 3, Stress levels end of the sprint (1-5)

Sprint/ Member	Pontus	Adrian	Jonathan	Lukas	Johan	Zakaria
Sprint 6	3	4	4	1	3	2
Sprint 7	3	4	3	4	3	3
Sprint 8	1	1	1	1	2	1

At the end of sprint 7, the developers with the highest stress levels at the beginning relaxed a little bit and some of the medium-lower ones increased, see Table 3. Likely due to the same reasons as previously mentioned.

Table 4, Individual satisfaction of sprint contributions (1-5)

Sprint/ Member	Pontus	Adrian	Jonathan	Lukas	Johan	Zakaria
Sprint 6	3	2	3	4	2	3
Sprint 7	4	4	4	3	4	3
Sprint 8	4	4	4	4	5	4

Thanks to a lot of features coming together and getting finished and probably a lot thanks to the updated and polished UI upgrade the team felt very proud at the end of the final two sprints, see Table 4. The application was now working and enough for a user to actually find somewhat useful or at the very least see the true potential in. Again this goes to show that

spending time and effort on a user-friendly interface makes a whole lot of difference for satisfaction levels.

Table 5, Individual story points (S = 1, M =3 , L =5)

Sprint/ Member	Pontus	Adrian	Jonathan	Lukas	Johan	Zakaria
Sprint 6 30p **(83%)**	L = 5	M = 3	M + S = 4	M + M =6	M + S = 4	M = 3
Sprint 7 40p **(82%)**	S +2M=7	M+M = 6	L=5	M + S= 4	L + S = 6	M + S + S =5
Sprint 8 25p **(88%)**	M+S =4	M+S = 4	M+S=4	M + S*X = 4 + ?	M = 3	M = 3

The tasks were assigned a size ranging from XS-XL though only S-M actually was used (beside Epic). All members made important and valid contributions as shown in Table 5. The team had a mixed background trying to play to each other's strengths. The third API used was the number of story points completed vs. undertaken. We set a general rule that if more than 80% of the team's total points undertaken were completed it was considered a successful sprint. All considered sprints fulfilled this KPI.

In future projects we would like to discuss early on not just what KPI:s to use because "you have to use them", but rather see them as an actual evaluation and reflection tool. The values should be developed in a way in which both the developers and the PO find them useful and are simple to incorporate into the general work cycle. For example, during weekly meetings. This doesn't have to be a long meeting but it should be a regular and central part of each sprint. They can be used for evaluating team progress, which is important to the PO, and team morale, to understand the team's health. Since projects normally span over much longer periods of time, we could also view KPI:s as a more long-term metric which sadly was not possible during this project.

The total amount of points undertaken vs. the amount completed KPI could quite easily be expanded upon. By including both team velocity and individual velocity this KPI could become more of a motivational factor both in terms of friendly competition both with one self and other developers. It is important that this does not become a method of poor criticisms that could escalate into persecutions.

To get more out of the reflections there must be a greater emphasis early on to incorporate the KPI:s into the sprint meetings in order to make greater reflections and analyze how the team is doing; both mentally and performance-wise. The team should discuss the KPI levels, for example, stress levels on an average level and if there are any outliers; why this is. For example, why did we end up feeling so stressed during week 7? Could we have done something differently earlier, catching whatever caused it? If the reason was only that it was

the last weeks it might be a good idea to implement more regular demos with each, or every other, sprint to make the effort for each sprint more motivational. This could be integrated in the weekly supervisor meetings or better yet, with the product owner. In other courses with longer spans this has been very effectful both in actual sprint performance but also trains the team in learning how the application should be presented to a customer, what the core values are, and how to sell it. The feedback from the PO or supervisor is extremely valuable to sort out priorities for future incrementations.

Social Contract and Effort

Your social contract

Our initial social contract defined the general rules and structure of our meetings and did not contain very much detail about the Scrum process. We had procedures in place for record-keeping, a schedule with a fixed larger meeting and at least two additional stand-up meetings each week as well as a clear definition of the expected effort and responsibility from each individual team member. As the project progressed and we started to get a better understanding of Scrum, we updated the contract with rules and a template to follow when working with the Scrum board. We also defined the workflow we had developed in a more formal way in terms of how we work with git branches, merging and reviews.

Having a detailed social contract was beneficial since it minimized misunderstandings, contributed to consistency thanks to well defined templates and forced us as a team to reflect on which practices to adopt.

Having a more detailed social contract from the start would result in the structure of the Scrum board to be clear and functional from the beginning making it easier for team members to see progress early on.

In a future larger project it would also make more sense to have more specific roles such as testers who primarily focus on reviewing and testing deliverables according to the definition of done.

Our meeting schedule with only one larger pre-scheduled meeting each week and at least two shorter stand-up meetings had the advantage of being flexible. But if our stand-up meetings were more strictly scheduled and evenly distributed throughout the sprint it would encourage team members to also distribute their effort more evenly throughout the sprint.

A way to reach a project where we have more specific roles could be by defining a set of roles which later on will be assigned to a team member, either as a fixed role for the whole duration of the project or on a rotating basis. What we could have done better is most likely more communication in order to know what each role’s tasks were.

The time you have spent on the course and how it relates to what you delivered (so keep track of your hours so you can describe the current situation)

Table 6, time spent

Hours/week	Pontus	Adrian	Jonathan	Lukas	Johan	Zakaria
Week 1	16	16	15	15	15	15
Week 2	18	19	17	21	19	17
Week 3	21	20	18	17	20	21
Week 4	20	19	19	21	17	24
Week 5	22	17	21	23	18	20
Week 6	23	22	21	20	23	21
Week 7	25	24	23	24	24	23
Week 8	16	19	17	19	20	18
TOTAL	161	156	151	160	156	159

As evidenced by the time spent, stress levels, and satisfaction levels discussed previously it becomes quite obvious that the early stages of the project were spent on learning how to use Agile and Scrum, the middle stages were spent working “regularly” and the latter stages more effort was required to finish. (Note that week 8 was only a half sprint but still had a relatively large time spent on it in order to finish). The hours we spent on the course didn’t always correspond to the progress we made on the product. As we just mentioned, much of the time we put into the course early on went into learning and understanding the different principles and practices of working agile with scrum. In a future project, if all the team members are experienced agile developers the hours we put in each week will likely line up better with the rate of progress of the product being developed. But even then the hours spent on each sprint can be misleading. Maybe a ton of time was spent on a particular sprint due to a serious bug. And maybe another sprint saw the biggest increment throughout the entire development process but the time spent was still below average due to great teamwork and high momentum.

To make the hours spent be a better indicator of valuable effort and progress, we should in the future make sure we are all on the same page as early on as possible with how we want to work and what we expect from everyone. This could of course be done via social contract and other forms of documentation but we also see a value for a new team to talk about their prior experiences and give time to get to know each other. This would lay a foundation for the team where measuring the time spent, per sprint for example, and reflect on how it relates to deliveries would be of much value.

Design decisions and product structure

How your design decisions (e.g., choice of APIs, architecture patterns, behaviour) support customer value

Early on in the course we established that we wanted to design the architecture of our product in such a way that made it easy to add new modules. This would work as a good foundation for allowing us to develop valuable increments each sprint throughout the course. Given that our product was in essence a type of calculator, we decided that the Model-view-controller, or MVC, software design pattern would be a great fit for us. In our implementation of MVC, the view was of course a graphical user interface that we developed using JavaFX. Every page corresponded to a .fxml-file where we defined and structured all of the interactive and visible parts of the application. The controllers consisted of .java-files used to dictate the behaviour and functionality of each page of the application. Lastly, the model mostly consisted of our database handler-module and the database itself. Our application didn't perform any remarkably complex calculations so a simple handler to insert and retrieve data was sufficient enough. This has worked well for us during the course because, as mentioned, it provided us with an easy way to add additional functionality to the application. There are of course other design patterns present in the application that increases the stability and quality of the product. We have for example used a singleton pattern for our database-handler.

In a future project we would much prefer to have both the idea for the application itself (product backlog), as well as our design ideas in place as early on as possible. For this project, we sort of worked on establishing the product backlog simultaneously as we were considering what design pattern would work best for us. A better way to do this would be to try to establish as much of the product backlog as possible in the beginning so that we have as much material as possible to base our choice of architectural models and design pattern on. This would not just affect the general approach of our work flow but would also allow us to maybe extend our definition of done and/or social contract to account for things like design patterns (which they currently do not). We of course don't want a waterfall approach where we feel like we have to finish the product backlog before we begin working on the actual product. Rather we want to allow ourselves the time to establish enough of the product backlog so that we can properly analyze what sort of software patterns and APIs could be applied to the product. The most important aspect of the design of the project is of course that we are all on the same page when it comes to how we should work to make it easy to add new modules to the code base.

Which technical documentation you use and why (e.g. use cases, interaction diagrams, class diagrams, domain models or component diagrams, text documents)

The technical documentation we used was a normal text doc, Class diagram, and an ER-diagram for the Database. All except for the class diagram felt relatively fine during work weeks and everyone felt at home using this technical documentation. However, looking back the documentation should probably have been more exhaustive in some areas. Especially

when evaluating team morale which was heavily criticized by superiors but got a lot better after some discussions.

In order to understand how our application would look like we wanted to map it out on a Class diagram (domain model) and the ER-diagram (for the DB). The class diagram was not well implemented and not really used. There were ideas for how inheritance would allow for more flexibility across the different categories but this was not maintained nor used to any particular extent. The ER-diagram was relatively well-updated as it rarely required much restructuring once the data structure was in place but when there was an update we tried to update the ER diagram prior to making any changes in the actual DB, see Figure 1. This was because changes to the DB would both affect multiple developer sections and not be detected by the version control like other updates would, instead it acted more as a binary detecting change or not. I.e., two developers working on separate things in the DB cannot merge. In the future we recommend having either a dedicated server for the DB or specific privileges for updating the structure of the DB so different developers do not work on features that might be overwritten during the same sprint.

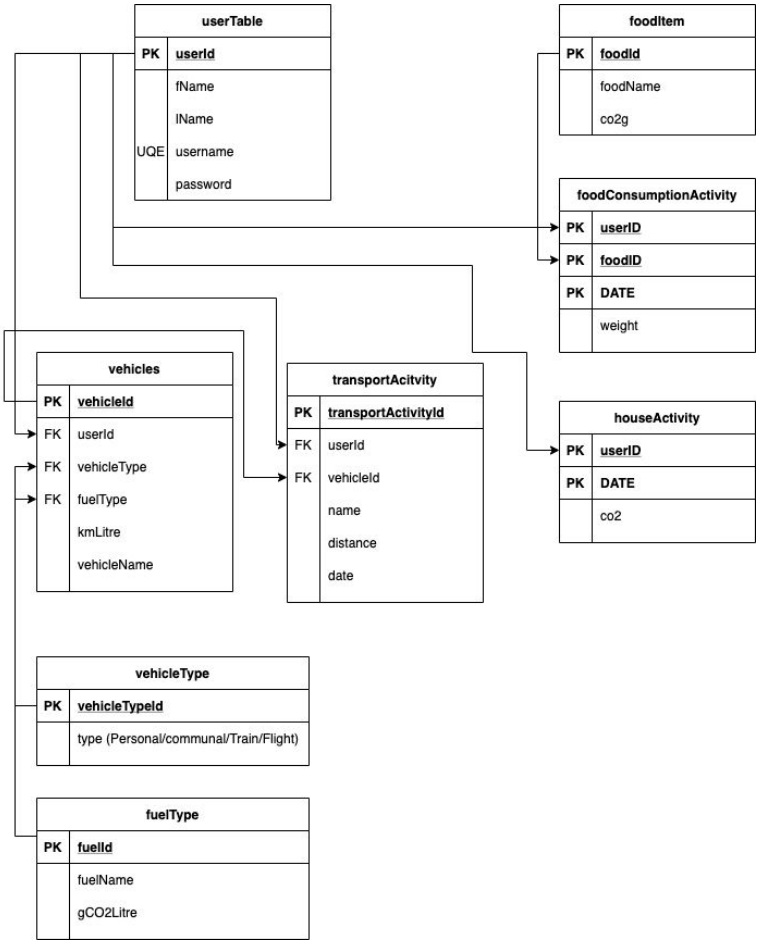

Figure 1, Entity relationship diagram

As a collective we discussed the main functions of the application, the end product could not be predicted by anyone in the group since we had different ideas of how the application would work and what functionalities we would like to focus on. By successfully adding more

classes and relations between the classes (and database) we further got a better understanding of how the end product would look like and where the project actually was heading towards. Using and maintaining a class diagram takes much time and effort and gives little to show for in a project as small as this. For larger more complex projects this would have been crucial and therefore a better system for maintaining it would have to be integrated into the workflow so everyone knows how the different sections should interact. This project should therefore, have been seen as a training for how to create and maintain such documents in a larger one. This could be integrated during the sprint planning meetings as it is during those meetings new features are discussed. It would therefore likely be the most natural opportunity to update how the interactions should theoretically look like while restructuring potential changes performed from the past sprint's implementations.

There were no interaction diagrams or use cases other than verbally discussed. This could have improved the feature implementations, especially those related to the DB implementations. This could have been implemented as a part of the definition of done for larger features, forcing developers to plan more before implementation.

As for the reflections we simply documented every week in the GitHub repository and it worked fine for us since it enabled most of the work to be centralized and easily accessible. The group documentation became more of a summary of the week's accomplishments and criticisms from the supervisors. It contained some reflections but lacked some depth. This could definitely have been improved upon using a standardized formalia and continuous updates after each meeting instead of once a week. The individual reflections were also documented on GitHub but we asserted that members reflections should remain private unless otherwise said. These followed a general structure and were great for reflections.

In a future project using improved standardized documentation is highly recommended since it makes it easy to follow the workflow and keep track of changes of ideas or issues regarding the project, but something to add or substitute would be the addition of JavaDocs. The reason is mainly to keep track of the code and have self-document so that no confusion arises later on and someone else could more easily take over from where you left off. Being able to pass on code without going through the trouble of explaining it would save time.

How you use and update your documentation throughout the sprints

During the sprints we documented everything in our weekly individual reflections and the team/group reflection. They contained documentation on how the sprint went, what to improve, and how to get there. For the code we simply put some comments in the IDE to create a reference for the future and shared our idea on how the different method worked during our "co-coding". During this time we sat together, shared our screens, and implemented the different methods together. Due to this and due to the fact that we had a short amount of time to produce a product we decided not to have much code related documentation beside proper code comments. The team and individual documentation was updated and pushed to the github repository at the end of each sprint.

For future projects it would help to have more elaborate code documentation. This would allow us to work in a more agile manner and take on tasks in areas of the code which we had not yet done any work in. We still tried our best doing this, but it was always hard digging into other people's code without more detailed documentation on how it worked. This could be improved by writing separate documentation for the different modules, their responsibilities and functionality etc. By using JavaDocs as a collective would fulfill the aspired documentation method for a future project along with updating the team and individual reflection as we did. This would also lead to a more agile way of working where documentation is accessible and understood by the whole team so that everyone can take on any task given to them.

How you ensure code quality and enforce coding standards

The code quality in our project has been governed by our definition of done. At the beginning of the course, our definition of done was a bit vague so we eventually decided to rewrite it to better coincide with the quality we expected from each addition to the code base. Our definition of done states that for each addition to the code base it has to be implemented, it needs to be tested and peer reviewed. We didn't prioritize high-end testing procedures so we decided that we were gonna stick to specification testing as well as integration testing. We expected everyone to test their own implementations before moving on to the next step, peer review.

The peer review consisted of someone else analyzing the implementation, providing a second opinion on whether the implementation performed as intended and if it was obvious to the reviewer. If it did not follow our definition of done or if there were other flaws, the implementer had to resolve the issue either alone or with help from someone in the group if it was needed.

Our domain model in this project was not completely implemented at the start which meant some classes did not have a structure to use. However the classes described in the domain model held a standard which meant that we had some parts of the code being based on it. This helped other team members see what their team did and made it easier to familiarize themselves with other people's code.

Another important part of ensuring code quality, which is also written in our social contract, was to write code according to the practises we have learned in previous courses on Chalmers. This included adding comments to your code in order to help other team members understand what you had done. Some of those things are writing clear code, implementing helper methods to improve readability and working object-oriented.

Code quality can always be improved, but one of the things which we could improve in order to make sure that our project followed a specific quality would be to test the implementation with scripted tests, using frameworks such as, for example, JUnit. These were not used as neither of the developers had previous experience with testing frameworks and it was not

considered a priority to learn testing in this course. Instead we focused on learning to apply Agile practices and Scrum.

Instead of taking a look at the implementation in the peer review, we could have looked more in depth of the code written by the team member asking for a review. By doing this we could take a closer look at what the member has written and judge their code depending on if they followed our guidelines according to the agreed contract/definition of done documents.

During our process of learning SCRUM we did not properly implement a domain model before we started to code. This led to some code being invalid / not being up to par with the updated model we implemented later in our project and therefore needed to change. In future projects more time could be spent on completing a proper domain model which will be followed strictly unless changes were required. By doing this a lot of old code would not need to be refactored and members of the team would also have a clearer goal to reach.

A way to improve the testing process would be by using Unit Testing for example where we could have set up automated tests which we ran in every branch to make sure it followed a standard. The reason why we did not put too much effort into testing in our project was due to the lack of- time and experience.

One of the easier things to improve would be the peer review since this is mostly based on how much time our team dedicated to it. If more time would have been spent we could have done an in more depth peer review. And if we also set a standard where we will control how strictly the code follows our standard we have.

Regarding the lack of a proper domain model there would be multiple solutions. We believe the best way for our group would have been to put more time into finishing the model before any code would be written. By doing this we can ensure that every team member at least has an idea of how most parts should be implemented.

Application of Scrum

The roles you have used within the team and their impact on your work

The three main roles of SCRUM, product owner, scrum master and scrum team member/developer, were planned to befall each and every one of us at some point during the project. The role of product owner has been shared by all. Seeing as it was part of the course that the team be its own product owner, we tried to figure out all of the most important details of our product early on so that we could fill our product backlog and get to work. Our first epics and user stories turn out to be less than optimal and were later reworked. Throughout the project, we decided if we wanted to change or add anything to our product backlog on our regularly scheduled meetings. The role of scrum master was, as mentioned, planned to befall everyone at least once, but seeing as there was not enough time, we just made sure as many as possible got the chance (we changed scrum master every sprint). The role of scrum team member was of course had by all throughout the entirety of the course.

The choice to have the role of scrum master rotate each week implied both good and bad. The good part was that most of us got the chance to try out the role of scrum master in a course on agile software project management. The bad part was that it was hard to properly learn to enact the role of scrum master in just one week (especially with no dedicated project owner). The experience may not have been entirely authentic either seeing as the scrum master is usually not also a developer on the scrum team. In the future, having a dedicated scrum master would certainly be better for all.

What we've learned in this course is something that is going to be applicable to a lot of different courses going forward. Given that this course was about learning how to work agile, we can see the value of sharing the role of product owner and let the role of scrum master rotate between us. In the future, however, we all think that a dedicated scrum master would be more than preferable. Hopefully, this will be a role many of us will feel comfortable taking upon ourselves (especially if others in a future project have no experience working agile).

To establish a strong scrum team in a future project, it will be key to make sure everyone is comfortable in their role(s) and that everyone also has a good grasp of the purpose of each of the other roles. If we find ourselves in a project where not everyone is familiar with working agile, a good start would be to educate the team. This will of course be of value even to those of us who have prior experience working agile seeing as everyone may not have the same experience.

The agile practices you have used and their impact on your work

We primarily followed the principles of SCRUM. We created a scrum board first on Trello but then moved to Github to centralize all the project related information in one place. In the beginning, there were many new ideas and practices to understand. We quickly understood the basics that there should be epic- and user story-cards and that these should be broken down and moved between the different columns, but in retrospect, we didn't understand how they should be formulated nor how they should be broken down. We mostly had the

perspective of developers when writing these which sometimes made the greater picture difficult to comprehend in a group. It also led to not breaking the user stories down properly into smaller tasks. We had many discussions with our supervisor who encouraged us to actually make the tasks into individual cards and move them around as he had noticed that our scrum board was rather static. After a few weeks, we, therefore, overhauled the board and made a clear document where we stated that epics are always issues, user stories are tied to epics (also issues), and tasks are tied to user stories (notes). Each user story has a priority, and each task has a size and is assigned to a developer. This made the board much more dynamic and "colourful" making it more motivational to interact with (moving cards is more satisfying than ticking a hidden checkbox).

There were some issues with linking cards which would probably have been very beneficial to restructure to get a better overview. Late in the project, we created our own separate column for user stories and tasks for the current sprint and something along those lines, but with better linking would greatly benefit us for a clearer overview and workflow. We tried to maintain this in the different iterative branches, but given that each sprint was so short it didn't make its way to the board properly. This would probably come in handy in larger and longer projects where tags for different categories (e.g. DB, statistics, etc.). This could make it easier for developers who are more comfortable in one field to filter the stories/tasks available in that field (given they had priority).

Each week represents a sprint, making them very intense. We used a rotating scrum master with the intention that everyone should have the chance to try the role, although time was not enough which meant some members missed the opportunity.. We tried to have three meetings every week with one of them spanning most of the day. This was originally on Wednesdays, but was moved to Mondays to better suit the supervisor meetings and sprint planning. This also became the unofficial "branch merge day". There were, of course, mergers throughout the sprint, but often these were smaller increments and not the larger sprint-into-master merge. When a feature was considered done by the developer it was pushed to its own branch with details of what has been done. A pull-request was then created and the branch was tested by another developer before it was merged (or dismissed) with the sprint branch. When there were actual conflicts these were brought to the teams attention and either solved during the big merger or the branch developers associated together solved them. In future larger projects where the sprints are longer this would probably have to be solved in a more structured manner. The optimal setting would be to have a dedicated tester doing two tests, one blackbox, representing how a user might interact with the application and one whitebox together with the developer or at least with documentation. Due to the short sprints given in this course the dedicated time of testing was too short and nobody really had a focused role of just testing the others code, instead we as a collective fixed both minor and major issues while screen sharing. Another thing to have in mind is that we come from different backgrounds and thus this leads to expertise being very spread and in the case of testing, all of us knew about JUnit which is a regression testing framework which is integrated with Maven (the project management tool we used), but not everyone knew how to utilize Junit to

optimize the testing of the code and this is one of many examples of how vastly different our competence was.

The sprint review and how it relates to your scope and customer value
We did not have any official product owner. Instead we mostly had early meetings where we decided what type of application and which functionality should be included and why. All the team-members, therefore, acted as unofficial product owners as they all contributed to the definition of the application scope. In the future more time should probably be spent developing what the minimum viable product should be, but leave enough room for expansion. This would require more exhaustive pre-studies and planning, but not too much as to remain agile.

At the beginning of each sprint there was a longer SCRUM meeting in which the developer team discussed what stories they had been working on, the core functionality of the new features and how they integrate with the project as a whole. This worked as an extended daily standup meeting with visual demonstrations. If the feature received approval it was merged into the sprint branch where some additional system integration testing was done. If it was not deemed accurate for what the team wanted it was either sent back to be re-worked and re-prioritized or scrapped. When the KPI:s became used around week 6 these were also integrated as a vital discussion point in the meetings to establish the team performance.

The definition of done was used as a document dictating when a user story was allowed to be considered done. This was used as a check liste for when an external tester reviewed a pull-request. If a pull-request did not meet the definition of done a comment was left and the task/user story was moved back from testing to in-progress until it was fixed and a new pull-request was created. This could have been extended to account for when a user story was allowed to move between the different columns on the board as well to provide even further details regarding development procedures.

Best practices for learning and using new tools and technologies
Only two developers had any practical experience with the GUI framework used. Therefore some of their knowledge was shared using a combination of tutorial videos and showing how certain important functionalities work and should be structured. Most of the communication was over Discord where different sub-servers were created for sharing different content. One was created for tutorials and one was created for help. When developers got stuck they could post a question and request help. Often these were solved using screen share, creating a sort of virtual pair-programming. These were of great help as the developer in need of help had to explain what he was attempting to do and in doing so had to think things through well enough to explain it to someone else.

The Scrum board saw a lot of changes throughout the project. At first the user stories were defined from a developer perspective which oddly made them harder to understand. Most of the user stories consisted of checkboxes which represented tasks that needed to be fulfilled for the user story to be considered complete. This was not very motivational to use as the

checkboxes were “hidden” and made the board look static as the card was only moved when all tasks were completed. This also made it difficult to keep track of which developer worked on which task and priority of the different tasks within the user story. The supervisor pointed this out several times and so we broke out the tasks to individual cards during each sprint planning. Later to get a better overview we created separate sprint backlog columns for user stories and tasks, that is, the relevant user stories for the tasks prioritized. Having them as individual cards made them more “fun” to use as the table became more alive and as a result became more utilized. There were still some issues with properly attaching tasks to user stories in a better way.

Nobody in the team had any real experience working with version control and github in teams. Most only had experience from individual projects. In the early phases it was therefore difficult to manage the conflicts as different developers worked on closely related sections. One of the big issues was using different java versions, especially when some different versions didn’t work with the JavaFx framework. We solved this by having everyone switch to the same version and editing it in the pom and iml files. As both the application and our knowledge grew these issues became more like interesting discussions and provided great insights in how the code actually worked and grew organically. When there were smaller merges with conflicts the developers responsible together solved the issues using screen sharing. Pair programming (even virtual) was proven to help many cases. This is a practice we will carry with us to future projects.

Relation to literature and guest lectures

The combination of having both lectures explain the core ideas of Agile and SCRUM and then ask specific project questions to the supervisor helped greatly with clearing up the confusions. One of the most recurring confusions was how to properly write user stories and how these should relate to epics and tasks. We believe that having only lectures and literature would not have yielded a proper understanding of how to use these frameworks and tools, which is why working with SCRUM in practise helped us. The group documentation reflects some of the criticisms/discussions of our supervisor, what the team felt about them and how to approach an improvement.

We used a handful of sources to better help us understand the art of working agile as well as related methods and technologies. The official scrum.org website was of great help for understanding how to integrate the theoretical frameworks into practice. The book Scrum and XP from the Trenches, 2nd edition, by Kniberh, H (2015) was good material for new scrum users and provided plenty of examples of how to go about working with scrum. When we realized that we had to restructure our scrum board during week 5 some of the ideas which came into discussions were influenced from the literature. The Pro Git helped clarify how the version control works and how to solve/avoid merge conflicts. There is still much to be learned as we mostly used it as a lookup for when issues came up and we didn’t know how to solve them.