Final Reflection

Group Jigglypuff

DAT256

Jimmy Andersson
Theodor Angergård
Jacob Eriksson
Martin Hilgendorf
Kevin Kullgren
Olle Lindgren
Elias Sundqvist

June 1, 2019

Customer Value and Scope

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

A: The app's purpose is to serve as a platform that connects two groups: people wanting to deliver stuff and people wanting stuff delivered. The deliverer category targets students, which we are ourselves, and therefore we feel confident that these users see an appeal with the app. From some group members' past experience working in retail stores we are also confident that there is a demand for cheaper transport solutions, which the app provides.

An obvious problem with the value proposition for the deliverers is that the money lock (i.e. the security deposit attached to a delivery) might detract deliverers, since the first thing they would have to do in order to make money is to deposit money into the app.. It does however help the parties trust each other which we think is one of the major problems an app like this would have to solve. Having to login might also detract users and make the value proposition harder to reach but secure authentication both helps the parties to trust each other and enables us to handle the transactions, which is our source of income. An alternative way of generating revenue would be to have one or both parties pay to use it but that seems even worse.

We have built almost all the functionality that we want and feel happy with the result. We wanted to include a map in the application, which would have given a better overview of nearby the deliveries but once we reached the point in the project where all the essential functionality was there, we felt that there might not be enough time left to create a well-functioning and bug-free map, so rather than taking the risk of spending a bunch of time on a map that may or may not work properly, we chose to focus on other functionality.

We had a few other ideas, which were deemed non-essential and therefore not included. These include subscriptions to transports, express deliveries which are more expensive, uber-like ratings for both parties and much more. Subscriptions would ensure there is always a certain amount of things to deliver on the app and there would be periodical revenue coming in.

B: Just like Kniberg says, planning a project so that a minimum viable product can be created as quickly as possible is a good idea. This MVP would have no functionality that is not essential and the later versions of the product would only have features that add significant value for a large number of users. If we can initially focus on only one group of users, we believe that that would make it easier to fine-tune the value proposition for their needs.

We would also like to have a good source of feedback on what does and does not add value and we would have a clear strategy for launching our platform from the beginning, taking into account how and when the platform would become profitable. The platform would preferably have strong network effects so that the value grows with the user base.

A → B: We think creating a good value proposition would be much easier if we had an external product owner that's not concerned with the technical difficulties in developing the app. In a commercial environment there would also be some sort of customer in mind that might be able to give us quick feedback, to help us understand if what we're doing adds value or not. Kniberg advocates short feedback loops which we think would help us achieve this.

We also think that successful platforms are easier to launch if you only have to focus on one segment of users initially. In our case we would have had to figure out some completely different value proposition in order to do that and therefore we chose to try and appeal to both sides from the start. For a platform like this one, with powerful network effects, going for scale before profitability would enable us to quickly improve the value proposition.

The success criteria for the team in terms of what you want to achieve within the project (this can include the application, but also your learning outcomes, your teamwork, or your effort)

A: The team's success criteria were initially having a balanced workload, no conflicts, learning new technologies and getting good grades. The team later refined these to include developing a complete application that delivers value to some customer, maintaining a good mood among the group members and working efficiently with a balanced workload.

The learning goals included generally the content of the course and specifically Git, Scrum, Kivy and Python early on. Later some group members formulated goals to learn to use Firebase.

There is no doubt that we underestimated the difficulty in learning new things. Both Kivy and Firebase took much more effort to learn than we had anticipated and therefore we only became productive by the end of the project. We sacrificed a lot of productivity in order to learn.

B: We would like to have clearly defined and easily measurable goals at the start of the project that reflect what all the stakeholders want and that the team are happy with and feel motivated by. That way it would be clear what our expectations are from the project and it would be much easier to work toward fulfilling those. If we have learning goals, we believe in learning by doing.

A → B: Having a discussion of what our goals for the project are and writing these down in a specific way will help us work toward fulfilling these goals and come up with related KPIs. Since learning isn't easily measurable, learning is not a good success criteria. If we want to learn new skills, a good success criteria to reflect that would be to have done something that requires those skills.

A good way to come up with success criteria would be to listen to the customers' requirements and formulate these as success criteria. If fulfilling those criteria require skills that the team doesn't have, they could either learn those skills or add a member to the team that already have the required skills. Since projects undertaken in a commercial environment always have a customer in mind, this seems realistic and if it's not because no one can figure out who would want the product, maybe the product shouldn't be created.

Usually there are also other stakeholders. These might include include the team itself but also investors, management, government, society and many others. We believe it is very important to figure out who these stakeholders are and what their requirements are. It is important here not to underestimate the significance of the team itself, as their wishes will reflect their motivation to work on the project, which is crucial to their success. As Kniberg says, motivation is more important than both effort, competence and environment for a team's productivity. Therefore the team itself should decide on a lot of the success criteria.

Whether it is acceptable to sacrifice productivity for learning will depend on the project but overall we believe in being pragmatic and having some purpose for learning things. If there is an easy way to do something, then do it that way.

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

A: Initially, we had a rough idea of what we wanted to do. The first week we broke down this rough idea into user stories, and then further into tasks which could be done during the sprints. Each user story had an associated list of acceptance criteria which needed to be met before it was considered to be finished.

We felt some reluctance towards assigning weights to the user stories initially. We had two reasons for this. First of all, since we were a team of people with different experiences and skill sets, the difficulty of a task could vary between each individual. Secondly, it was quite difficult in general to assign weights to things before we had familiarized ourselves sufficiently with the tools and technologies involved.

However, after the first week we realized that this was a mistake. We intended for velocity to be one of our KPIs, and this was impossible to measure without having assigned weights in the first place. So we assigned weights to both the user stories we had completed, and those that remained for the first sprint. For all following sprints, we made sure to estimate all tasks from the beginning, and added them to the sprint based on what we expected to be able to get done.

We did not use any special method/standard pattern for the weight assignment. The basic principle instead was to evaluate how similar in difficulty we expected each new task to be in comparison to the ones we did previously and then assign a value that was approximately an average of what the group members estimated.

Initially, both the estimated weight and velocity was very rough and approximate, but each week we strove towards having a more accurate estimate of velocity. Each week except the last we somewhat overestimated the amount of work we would be able to do, but the estimates became more accurate over time. We had two weeks with a consistent velocity of 50 points. The last week we achieved ~80 points, but that involved many team members working overtime. So our true velocity for a normal week seemed to converge to around 50.

While velocity and weight estimation initially, to some of us, felt like a too difficult and uncertain task to actually be useful, this turned out to not be the case at all. By simply following the process we eventually converged to something that could be used as a practical and efficient tool for predicting how much could be done each week, and more easily prioritize which tasks needed to be done from a value-oriented perspective.

B: For future projects we would like to use user stories and weighted tasks to keep track of what needs to be done, how much work is required for each of those things, and how much work we can expect to perform over a given period of time. We want to use this as a tool for decision

making about where resources should be allocated, primarily for the purpose of creating value for the stakeholders.

A → B: Since our approach to user stories and task estimation seemed to work out in the long run for this project, we will most likely use a similar approach in future projects, the biggest change that we would like to make is to estimate the task weights already at the first sprint planning. We would also like to start off with underestimating the velocity. It's more motivating for the team to see the velocity KPI slowly rise, than the other way around.

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

A: In the beginning, we struggled a bit with understanding who our stakeholders were. Since we didn't have an external product owner, we felt like there was no one who would evaluate our work from that perspective. This lead us to start out with less focus on what the product owner might find important, and focus more on the parts that we, as developers, felt was important. A lot of the feedback we received in our meetings with the instructors confirmed that this was actually the case, and we got to hear the phrase "you are discussing details instead of looking at the big picture" quite a few times. After our first sprint, we appointed Olle and Kevin as product owners. This made it easier to decide priorities for the user stories. Another problem we ran into was the fact that we didn't have any real customers that we could ask for feedback on our ideas and solutions.

On the other hand, we had a pretty clear idea of how we wanted to lay out our acceptance tests from the developers' point of view. Early on, we decided that no code would go into the main code base if it wasn't properly reviewed by at least 2 members of the group that had not been participating in writing it. We also set up an automated pipeline to help us test code coverage, code style and documentation, all of which has to pass in order for a developer to be able to merge a new feature into the main code base. These guidelines proved to be useful, as the team felt that they could rely on a process that ensured high quality code and documentation of the application, and this made it easier to get acquainted with new any features that were added throughout the project. These routines were kept for the majority of the project, the exception being an unwieldy feature concerning user authentication, which after some thought, discussion and a lot of extra manual testing was granted merging rights despite the code coverage being less than perfect. However, the code coverage was fixed in the following days, as that was one of the conditions for it being merged into the main branch.

B: For future projects, we want to have a well defined idea of who our product owner and our customers are. We want to make it easy for ourselves to reflect on decisions and deliveries from the perspectives of those roles, so that we can be sure that we are solving real problems, and not simply solve what we think might be the problems. We would also like to automate as much of the technical acceptance testing as possible, so that the manual testing can focus on the practical functionality, user experience and whether or not it solves the intended problem. An intended problem could for example be a common user flow in the application.

A → B: By making sure that we have a product owner (or at least a team member who will take on the role in review and evaluation situations) from the start of the project, we will be better equipped to handle the outlining of requirements, the evaluation and prioritization of deliveries.

We could also recruit potential customers that we could use as sounding boards for our ideas. This could be friends or family members to start with. By asking questions about what problems they would like to be solved and how they feel about using certain solutions, we will be able to better understand what is important for the end users and to better define relevant acceptance

tests. This will take much of the guesswork out of the equation for the developers, as we will have much more information to base our definitions of done on.

By keeping and refining the CI pipeline, we can make sure that many of the technical aspects are reviewed by automated systems, freeing up more time for testers to focus on whether or not the changes adds any value from a customer perspective. One such improvement could be to place the backend service under continuous integration as well. Since Firebase was used in this project, we did not have the opportunity to do so, but it would have freed a lot of time since any change to the database structure was likely to cause problems in others branches until the proper updates had been implemented and merged.

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

A: During the development of this project, the team used three KPIs: sprint velocity, code coverage, and weekly stress levels aggregated from daily scrum using a scrum bot.

Sprint velocity

At the start of each sprint, the team picked the user stories that were supposed to be implemented during the sprint. We assigned a weight to each user story, and the sum of these was our estimated sprint velocity. At the end of each sprint, we calculated the total weight of the user stories that we had completed to find our actual velocity for that sprint. The estimated and actual velocities for each sprint are shown below:

Sprint	Estimated Velocity	Actual Velocity
1	250	130 (65/week)
2	70	50
3	65 + an unknown amount	50
4	55 + two unknown amounts	~80

The first sprint was 2 weeks long, in order for the team to get started learning the required tools, technologies, and processes to start developing for our target environment. Due to the long sprint, we decided on a relatively high estimated velocity. However, at the end of the sprint we had only achieved about half of that – we had drastically overestimated the amount of work we could complete while also learning a lot of entirely new libraries and frameworks. Thus, we reduced our estimated velocity to 70 for the next sprint, thinking that we had learned most things we would need and development speed would pick up. However, in the end, the actual development velocity was still lower than what we had planned. Using the same reasoning, that the development pace would pick up as we learned more, we allotted 65 points worth of issues for sprint 3. Here, we also included an issue regarding fixing APK builds that we were not able to quantify in advance, so we left is as an unknown amount. This issue alone ended up costing 2 team members at roughly 65 man-hours that week. After giving up on the issue, development continued as usual, but we only completed issues worth 50 points. For the last sprint, we picked and prioritized issues that would give direct value to the demonstration of the product, which also included two unknown issues: completing the Firebase backend configuration and client side code, and getting a location API working. However, this sprint we were successful in completing all the tasks and even got around to implementing some more fixes and polishes than originally planned.

Code coverage

Code coverage was our least utilized KPI for this project. CI and codecov checks where set up before the first sprint began, but there was some confusion regarding how the coverage

reporting and coverage status worked. We also did not utilize test driven development (TDD), and because a lot of the backend logic was implemented on Firebase using functions and rules, writing unit tests was not a common tasks for most issues that were implemented. Thus, we did not really write many unit tests until the 3rd sprint. Until then 3 dummy/example tests had provided 88% coverage of some example code that was removed during sprint 3, causing coverage to drop. From this point, we started writing tests to achieve maximum coverage, which finally landed the test coverage at 93%.

Scrum responses

The Slack bot for our daily scrums has a feature to download all responses as a JSON file, which was very useful when analysing the responses. We included a question to quantify every team members daily stress level, which allowed us to calculate the following average stress levels for each sprint:

Sprint	Average Stress Level
1	3.7
2	3.5
3	3.6
4	3.6

The variances on these values are quite high. There were several days where one or more team members reported a perceived stress level of 5 (the maximum) while others felt a lot less stressed. We utilized this KPI both on a day-to-day basis and every retrospective. Daily we helped other team members out if they were too stressed with their work or others who had little to do. During each retrospective we briefly discussed the stress levels that we felt throughout the sprint, and came to the conclusion that we were fairly happy with the stress level and that we wouldn't need to take any action. Below are the average and median stress levels per person:

	Martin	Kevin	Elias	Jacob	Olle	Jimmy	Theodor
Average	3.36	3.71	2.96	3.13	3.19	4.00	4.57
Median	3	4	3	3	3	4	5

B: For future projects, we want to choose relevant KPIs that help to evaluate the success of the team and project/product. They should be chosen carefully with good understanding of what is important for the project. Doing this will allow the team to effectively utilize the KPIs in order to identify potential improvements that should be made to the process or product.

A → B: In future projects we should make use of the data from tests coverage right from the start, and have “code coverage must not decrease” as a mandatory requirement for PRs. In combination with utilizing TDD and having rigorous guidelines for writing tests from the get-go, this would help to ensure that high quality tests cover as much code as possible. Along with requiring all PRs to not reduce the code coverage, this would ascertain a good code quality.

Social Contract and Effort

Your social contract, i.e. the rules that define how you work together as a team, how it influenced your work, and how it evolved during the project (this means, of course, you should create one in the first week and continuously update it when the need arrives)

A: The team wrote and accepted a social contract during our first meeting, so that rules and expectations were clearly laid out and everybody knew about our ambitions and goals. This initial contract was written based on different social contracts that we had used in previous group projects, where they had already been extended and amended to suit our needs. We re-used the social contract from one of the I-students previous project groups, and extended it with a few clauses from the IT-students' previous social contracts.

In our social contract, we specify how communication is to be handled, how decisions are made within the team, expected working hours/procedures, and how conflicts, absence or tardiness should be handled. The contract also specifies the attitude towards the coursework and the expectations on effort; all members are aiming for a high grade and their work ethic should correspond to this ambition.

The team only revised the contract once to remove the role of meeting secretary. This was because the people who came to the meeting first, usually ended up being the ones taking notes anyways. Overall, the team was happy with the social contract. No problematic situations ever occurred, and because everybody had agreed to the social contract at the start of the project, absences and late arrivals were handled accordingly and to everybody's content.

B: A fully functional, tried-and-tested, completely covering social contract that aids in the teams process and defines processes and solutions for all kinds of situations. The contract should function both as a set of rules for the group to follow and as documentation for how the team wants to operate, and why.

A → B: There are some more theoretical situations that could be covered in the social contract such as how the group wants to handle upcoming deadlines, how tasks are assigned, and how decisions should be made in case of stalemates (in case there is an even number of team members; otherwise a stalemate cannot occur anyway). Previous social contracts used by IT students also included more detailed requirements about implementations and coding requirements, which would help to create a more full-fledged social contract for a software development team.

The social contract should also undergo revision and amending during the course of the project work, to ensure that it is always relevant to the team's expectations, ambitions, and processes.

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)

A: Throughout the entire course, we aimed for all group members to have an average workload of 20h/week. And this was also approximately what ended up happening. Of course, since this is real life, the amount of time that has been available to the group's members have varied between the different weeks, but we have collectively tried to compensate for that on other weeks in order for the average to be maintained. Through the daily scrums, we have been able to know when the time has been limited for the different members, and plan our work schedules with that in mind.

Towards the end of the course, things became more stressful however. As the presentation deadline approached, the members felt the need to work overtime in order to finish everything needed for the demo.

There were several time sinks that slowed down development. The key culprit for most of them was our choice of backend, Firebase. The problems were fourfold.

- We had no way to version control the backend, so older versions of the app eventually became unusable, and it became difficult to work on some things in parallel.
- A large part of the functionality of the app hinged on parts of the backend working, so it blocked many of the tasks being worked on and decreased overall task independence.
- Getting authentication to work took several weeks and did not provide a value proportionate to the amount of work invested in that particular feature.
- Some of the dependencies for the firebase API made it impossible to build the APK for the app, since python-for-android doesn't support the grpc package yet. Close to 65 man-hours were spent trying to fix that issue, since app builds are crucial for the project, but even with the help from the devs of python-for-android we ended up having to give up on APKs.

All things were not so bad however, the choice of using python meant that very little code was required to implement functionality. Furthermore, the component library KivyMD meant that we could effortlessly add good looking graphical components to the app, which saved us a lot of time on graphical design.

With that said, we did get a lot of things done, all members have learned new things about agile development and the workload has at least been manageable for the most part.

B: We would like to spend a fixed number of hours per week in the next project, keep stress and workloads manageable, while simultaneously making steady progress towards achieving the success criteria for the project and the team. We would also like to completely avoid time sinks, or at least identify them quickly enough so that the approach can be changed without incurring

prohibitively large setbacks. We would also like to be better at creating and identifying tasks which are cost effective in terms of provided value per man-hour.

A → B: Having a balanced workload in terms of work hours per week was a good method to mitigate the risks of stress and burnout, while ensuring a steady rate of progress on the project, so that is something we would like to keep for future projects.

This would have worked close to perfectly in our minds, if it weren't for the different time sinks, which in fact made some members feel burned out.

In order to prevent time sinks in the future the sprints should be short, especially in the beginning, so that different ideas can be proposed/explored/tested quickly before too much time has been invested into any particular one. This should give a more concrete feeling for what might be time consuming compared to just discussing things based on how they are presented online, which we did a lot of this time.

Design decisions and product structure

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

A: It was important at the start of the project to find the languages and tools that would allow us to quickly produce something that would give good customer value. Because of this we decided to use the Python language together with the Kivy framework. Python is typically simple to read and quick to write, and with Kivy we could run our application cross-platform. This meant that we could export our application to both the iOS and Android market, while still running it natively on any desktop.

Kivy comes with various components which made prototyping quick and easy. However, the default components are built from a functional standpoint, not a UI/UX standpoint. Therefore, we also started using KivyMD. It's a library that is built for Kivy and contains a lot of components and different stylings. Its components strives to follow Google's Material Design specification, and this saved us a lot of time in providing a professionally looking application with good UI/UX for our customers.

As earlier specified, one of the benefits with using Kivy was the ability to export to both iOS and Android. This was only half possible for us. Due to technical difficulties related to the python-for-android project, it was not possible to export the application to Android. This meant that our final product could only be run on the desktop and iOS. This is a great loss of customer value, since it's not easily accessible. The best solution to this would be to pay the python-for-androids devs to add support for the grpc package, which could likely be done within 1 day, and thus wouldn't be expensive.

We decided in the early stages of the project to use Firebase for our backend, database, image hosting and authentication. One strong reason for using Firebase was that two members of the group had previous experience with it. The plan was that Firebase would give us a great infrastructure, but due to the Python support being buggy and the documentation misleading, integrating it into our project took more time than expected. Time that instead could have been spent on other features that didn't make the final product.

We used Travis as our continuous integration since we have used it in previous projects, has good documentation and is free. This helped us keep the code documented, properly styled and have a less buggy code via tests. We also used codecov to monitor how code coverage changed throughout PRs. The end result was that developing new features took shorter time since we kept emanating from a stable code base, which in essence translates to a lower technical debt for the project.

In our application, the user is required to enter valid addresses. To keep this to one input field, and not having to enter postal code and city separately, we're using autocomplete suggestions

via the Bing map API. This results in faster creation of delivery requests which favors customer value.

B: Going forward, we want to make design decisions that has ground in both giving customer value as well as making development as easy and painless as possible. Using technologies that one or more in a team is familiar with can result in fast progress for the entire team. However, this thinking shouldn't rule over everything else. If a library is no longer supported it shouldn't be used in a new project. Even though the whole team might know it.

A → B: For the next project we need to dare to take extra time to make good design decisions. In the end, it should lower the risk of having to put up with a bad api or having to rewrite parts of the application. We need to be sure to always reflect on what we're putting our time into and how that's going to reflect in customer value. Because it doesn't matter for the customer if we have will use Firebase or Azure, it matters if it actually works, and that it works well.

Which technical documentation you use and why

A: We wrote our user stories in one document which our product owners then continued to iterate on. During each sprint planning we then picked user stories from the product backlog and converted those to Github issues in the form of tasks. These issues were then placed inside Github projects (one for each sprint), which are configurable and automated kanban boards. We setup ours so that we had the columns “To do”, “In progress”, “Review in progress”, “Reviewer approved”and “Done”. Things go from “reviewer approved” to “done” when it passes CI and the code is merged. Since every action in this ecosystem is automatically tracked we get full documentation on our entire workflow, and thus it serves a documentation for the development of the codebase.

We were quite consistent with writing formal and detailed tasks during the first 3 sprints, but we ended up cutting some corners on the last sprint with issue description quality. This was partly due to consensus in the group of knowing what was left to do, we just needed to get it done.

The code has been fully documented (using Sphinx docstrings) during the development lifetime. This made it easier to understand why/how code worked, and made reviewing easier and faster.

We wrote user flows to formalize the most common important sequences of actions that our clients wants to be able to perform in order to gain value from the service. This helped us understand if our designs and implementations were sane, and also helped us in focusing on testing real-life scenarios that would give our clients value.

We made a mock to internally decide on the design of the app, and updated it until the design was implemented. Using a mock was very useful for agreeing on a design, spot design mistakes early on and we could use the mock as part of our definition of done when reviewing any visual implementations or changes. By the end of the project it got a bit outdated though.

We also wrote specifications for the data storage structure we used on Firebase. We did this in order to agree on a structure in the team before implementing it. It could also be used as a reference to check where certain values where stored and what data type it used.

B: All modules/packages/directories, classes and functions should have documentation describing their purpose and how they achieve that purpose. This would ideally make it possible for any team member to understand, maintain and develop any part of the codebase. This would help form T-shaped developers, result in a high truck factor for the group and minimize time spent questioning how/why code works.

Furthermore, the design of the app and the interaction between code modules, using mocks and component diagrams respectively, needs to be clearly visualized and constantly updated. Both are for detecting design errors before implementing them, and also to make reviewing any implementation easier. Visualizing interaction between code modules will also make it easier for developers (especially new ones, if any) to understand the system as a whole.

If the next project is a bit longer then a paper prototype is desired. Paper prototypes take some time to develop to a useful degree since they need to be interactive, but often end up saving time in the long run. This is because they both serve as documentation, and a way to test the potential app with real customers early on before anything is implemented - giving us early feedback on design and features.

A → B: We documented classes and functions and ensured that through CI, but we didn't cover documenting each directory (module). Our code would likely have ended up more modular if we would have done this, since it would become apparent from the module description if the module was too big/broad/vague. Therefore, we should enforce module documentation in CI.

To clearly define interaction between modules we will either write simple component diagrams that focuses on dependencies, or use a program to auto generate UML from source code. Both of these approaches would get us close to B and would not be time consuming to maintain.

To keep the mock updated we should add to our definition of done that "if there are any visual changes in a PR that diverges from the mock, then that PR must be either rejected or the delivery team together with the product owner must agree to change the design in the mock to match the PR" or something similar.

How you use and update your documentation throughout the sprints

A: We adopted a healthy culture of updating and creating documentation as soon as it was needed from the get-go, which made our work much simpler in many ways. We made sure to take notes of any important information during our sprint meetings and retrospective meetings, including what we wanted to achieve, what was and/or wasn't achieved, and the reasons for reaching and/or not reaching our goals. This made it simpler to focus on things that didn't work as well as we wanted them to, and to discuss possible improvements to our processes. We also documented our process and progress in our team reflections and in our individual reflections.

Any decisions that were made regarding what tasks to focus on was documented on our project boards on Github. Discussions that formed these decisions was kept on Slack as to not clutter the Github threads. This made it easier for us to get an overview of what the team as a whole was working towards, how far we had come, and who was in charge of what.

With the help of Geekbot, a scrum bot integration on Slack, we managed to make our daily standups asynchronous. This made the standups easier to fit into everybody's' schedules since some group members woke up at 5 AM while others went to bed at roughly the same time. Since these stand-ups were in written form they ended up also acting as daily progress, status and health documentation. This made it easy for all members to keep up with one another and plan their own work accordingly.

B: In future projects, we would like our documentation to be continuously updated and readily available for everybody on the team. We would like the daily standups to be in a written form and so that they can both act as documentation and as sources of KPI data. That way, it will be easy for us to assemble the indicators we need to reflect on our work. We would also like to use templates for both sprint reflections and sprint reviews, so that we have a clear picture of what we need to cover each meeting.

A → B: By thinking about what relevant KPIs we want for the next project, we can design our standup questions in such a way that it's easier to automate the extraction of relevant data, and the calculations and visualizations of indicators. We should also create templates that can be used to make the meetings more efficient by including topics we need to discuss at every occasion, and make sure we don't forget any. The templates should also be designed in such a way that they can support the collection of any indicator data that we may want to keep track of.

How you ensure code quality and enforce coding standards

A: First thing we did was to lock our main development branch "dev" as a secure branch. This meant that no one could push to this branch - all code had to get there through PRs. Therefore, all the features and fixes had to be written in separate branches. This was nice since you could easily see what code was currently being written, switch between branches to test code and give feedback, and avoid “toe-stepping” and thus making use of force pushing and rebasing.

Forcing code through PRs also meant that we could filter all code entering "dev" by a set of rules using the education version of Travis CI. The CI rules we used for PRs were the following:

- needs to be rebased on latest "dev" commit and have at least 2 approvals
- needs to pass YAPF (PEP8 standard code style)
- needs to pass Prospector (collection of several static analysis tools / documentation)

Reviewers used our Definition of Done to determine if a PR would be accepted. The CI was also meant to require tests to pass. However, near the end of the project we realized that CI never actually complained if the tests failed - it just ran them and ignored the output. If we had noticed the problem earlier we would have gotten around to fixing it and saved some confusion.

Each PR showed how it affected the code coverage. We did not enforce that all PRs must affect codecov non-negatively, but encouraged it. When we first added it we did not really understand exactly how its reporting worked so we did not mark it as mandatory. We never got around to changing it, so reviewers had to determine what kind of codecov diffs would be reasonable.

B: In future projects, no broken, buggy, untested, undocumented or inconsistent code should be able to be merged to any main branch. Moreover, all PRs should only contain code that is short, concise, relevant and where every part of it contributes to some customer value. Despite all of this, it should still be quick and seamless for developers to get their code merged, so that efficiency is not hurt because of rigourous CI / quality checking.

A → B: Broken/Untested code and the second sentence in B are the main topics that need improvements. Broken/untested code can be improved to a satisfactory level by enforcing 100% code coverage in the CI rules and fixing so CI complains when tests fail. Furthermore, guidelines should be added for how to write tests in the Definition of Done. These guidelines would cover naming conventions, assertions per test and what reasonable edge cases are.

It is hard to tell a computer what short/concise/relevant and customer value really is, but as we see it there are 2 main approaches on how to improve on this topic.

1. Force PRs to a maximum amount of commits and lines changed. (well-defined)
2. In our Definition of Done clearly state what these values mean. (not bulletproof)

Merge time could be further improved by investing in paid Travis CI for parallel builds.

Application of Scrum

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

A:

Product owner

Initially, we did not have a formal product owner. This decision was based on the fact that there were no real stakeholders nor any real customers in this project. As the project started, the whole group determined approximately 40 user stories and tried to prioritize the most important and valuable features.

As the project went further, we decided to appoint the two industrial engineering students as product owners. However, they did not represent the bridge between any real stakeholders. A contributing factor for assigning product owners, except the fact that it was mandatory, was to get more experience with the scrum framework and stressing the focus of targeting customer value. The product owners made an effort to focus on the business side of the application and had the final say whether features should be implemented or not.

Development team

The group consisted of 7 developers, 2 of which were product owners since everyone had to code. During every sprint planning, all the selected tasks were divided among the group members. In the beginning of the project, people were not assigned any specific development roles. However, as the project went further, people naturally gravitated to certain areas such as CI, Firebase, and front-end development. For that reason, people ended up having more defined roles, and issues ended up being passed to the person who was the most efficient within that area.

Scrum master

Since we had no formal scrum master, the whole group shared the responsibility to clear obstacles as well as ensuring that the scrum framework was followed.

B: For a future project, the group members want to get more experience in the different scrum roles. The group also wants to learn how it is like to work with real stakeholders as well as having a scrum master that is responsible for the process of maintaining and promoting scrum.

A → B: In order to get more experience in the different scrum roles, people will have to participate in new software projects and try out new roles. To learn what it is like to work with a scrum master, one person in that team should be assigned the scrum master. In another project, the group could also search for real stakeholders who they feel would make the experience more tangible and closer to reality. There should also only be 1 product owner in the team, as per scrum's definition, and that person shouldn't be a part of the delivery team

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

A: The main agile practices we have been using in the project have been:

Iterations: We have worked in iterations over short sprints. This has allowed us to evaluate our progress, and to redirect our focus to ensure a finished product that delivers value in time.

Customer-oriented approach: We have tried to keep in continuous contact with the project owner to ensure that the product we created provided value for our intended customers, and we have redirected our focus and re-prioritized accordingly. Initially we didn't have a product owner, which made this difficult to put into practice. After the first few weeks however, we decided that the Industrial Engineering students should be the project owners. Since the project owners were part of the team in this case, this practice became trivial to incorporate, but some of the value of getting an outside perspective of the product was lost.

Product backlog: As product backlog we used a google docs document to keep track of all our user stories and their weights. For each sprint we also had a sprint backlog (a TODO) column in our GitHub projects' kanban boards. This allowed us to divide the large task of creating the entire application into smaller, more manageable subsets of tasks for each sprint.

User stories: To specify requirements for the app, we used user stories. These were a helpful tool for seeing what needed to be done, why it was valuable, and who it was valuable for. These were broken into smaller tasks that could be completed during the sprints.

Daily Scrum Meetings: We used a bot in the project slack workspace which helped us remember what was done the day before, our current goals and possible obstacles. Furthermore we measured stress level during these daily scrum meetings and used an aggregate of these as a KPI metric.

Sprint retrospective and review: We did not have a very clear separation between these two. We met once every monday, began our meeting with the retrospective and followed up with the review. Both provided their own distinct value, however. The *retrospective* allowed us to reflect on the *process*; we discussed how we could increase development efficiency etc. For the sprint *review*, we discussed the *product*; did we complete the tasks for the previous sprint, how did it go, where should we go next?

Continuous Integration and automated tests: We set up a continuous integration environment with travis from the beginning. This allowed us to continuously verify code quality and ensure that nothing would break when new features were added.

Velocity: We assigned a certain weight to each task, and kept track of our current velocity for each sprint. We were also able to see the current sprint progress by using the github project board.

Requirement prioritization: Using feedback from the product owner we prioritized the different user stories/requirements based on how much value they would provide compared to how much work it would require. The same principle was also used when selecting which tasks to work on within a sprint.

In-pairs programming: This agile practice was not used extensively, we did however have a few pair programming sessions to help the team members less experienced with programming get started. It was also used on some of the bigger tasks in order to distribute the workload more evenly. .

B: For future projects we would like to utilize agile practices to their full extent in order to efficiently and iteratively develop a product that provides maximum value to the product owner.

A → B: We have already used a number of agile practices extensively in this course, but there is always room for improvement. In the future we will try to separate the sprint retrospective and review more clearly, since the review in our case often ended up dominating the discussion and leaving less room for development of the agile process.

There were a few notable agile practices that we didn't use, and would like to use in future projects. One of these being test-driven development. Since the bulk of our attention was initially dedicated towards learning the new tools and getting things to work, writing tests became a low priority, and we argued that it would save us work if we needed to switch tools or architecture in the early stages. But later, since we already had untested code, it was difficult to enforce that all code should be tested. We did use tests, but the tests were never written before the functionality. In the future we would like to do this, so that we can always be sure that both new and old code is continuously tested and works as intended. We believe that the extra work required for writing tests in the initial stages will outweigh having to enforce test-driven development later on.

Another area of improvement for future projects is the use of burndown charts, this should help us more accurately track the progress of the sprints over time and potentially improve our velocity.

The sprint review and how it relates to your scope and customer value (in the first weeks in terms of the outcome of the current weeks exercise; in later weeks in terms of your meetings with the product owner)

A: In the first couple of weeks, the whole group took part in the puzzle and Lego exercises. Those exercises helped us understand the basics of working in an agile manner as well as the importance of delivering value to the end user. They also gave us an initial experience in making incremental improvements and in evaluating our process as a group. Another thing that we learned from the exercises was the importance of not making any assumptions. This was clearly demonstrated with the puzzle and the first Lego-sprint, where incorrect assumptions had a significant negative impact on everyone's work.

After our first sprint, which lasted two weeks, the time had come for our first sprint review. The first thing we realized was that we greatly overestimated our velocity. A contributing factor was that some people underestimated the challenge of learning a new programming language. Another issue was that people had less time than planned and desired. To improve the quality of our sprints, we took three actions.

- We lowered the estimated sprint velocity and tried to prioritize features that had a bigger impact on the project demonstration.
- To bring a broader perspective to the user stories and their customer value, the group assigned two product owners.
- To get more opportunities to review our process and reflect on what we had accomplished, we changed our pace from two-week sprints to one-week sprints.

We usually ended up delivering below our estimated velocity, but as the project went further our estimate got closer to our real pace. There was usually a different matter every week causing this; e.g. one week it was issue dependencies, another it was lack of time.

In the final weeks of the project, the group really stressed the importance of creating value and taking an end-user perspective into account. For that reason, we ended up cutting out some features and mainly focused on ensuring a viable application for the project demo. We also removed or explained any element that seemed unclear from a user perspective.

B: For our next project, we want each sprint review to conclude that as much executed work as possible brought value to the application. In every sprint review, we want to identify what went well, and what did not, in order to share problems and solutions to increase our productivity.The group wants to work with real stakeholders who can give us guidelines for our work. The group also wants to make good and realistic estimations of the group's pace.

A → B: In order to execute more work that brings value, the group will have to pay even closer attention to the stakeholder's and product owner's demands. Reflecting about the work and

having continuous meetings will help the group to get more experience in estimating their own pace and how to improve productivity.

Best practices for learning and using new tools and technologies (IDEs, version control, scrum boards etc.; do not only describe which tools you used but focus on how you developed the expertise to use them)

A: Throughout the project we used PyCharm or Emacs as our IDEs. For version control we used git and Github. We also used Github for our backlog and used their project boards as our scrum boards. A majority of the group members have worked with these or similar tools before, but a few weren't familiar with these tools. Throughout the project we helped each other learn the tools so that they didn't become obstacles, but rather accelerators to help us excel in developing the project.

Python, Kivy/KivyMD and Firebase were vast obstacles that few of us had previous experience in. But for every tool and technology there was usually someone who had previous experience with it, which meant that this person could setup the technology or help the rest of the team do so. Either via Slack, in our weekly longer meetings or when sitting together and doing pair-programming.

Something that was of great help was the Kitchen sink demo. A large Kivy/KivyMD application where many features were shown. Throughout the project, this was used as a great inspiration and help when working with Kivy and KivyMD.

B: It's important to find the major technologies that's going to be used early in the project, and sustainable ways for the entire team to learn them. The velocity of the team will slow down if there's uncertainty or confusion about the technology. Therefore it's important to have ways to help each other.

For the next project we should start off using mostly familiar tools to quickly get the team comfortable with the development environment. After that, we can start to introduce new tools and evaluate them over time. To keep a groups morale up, it's important to always feel like things are moving along. Having to read up on a multiple libraries as the first thing in a new project might be like running into a wall.

A → B: It's important to start with familiar technologies. Especially when there's a short time limit as we had. We did this to a reasonable degree but there were a few surprises which we didn't account for. So next we will more thoroughly investigate the tools we plan on using before actually integrating them into the project. We will also be focusing on familiar technology so that we can gradually expand and learn new technologies from a stable and comfortable base.

We should try to look for and use demo projects like the kitchen demo as much as possible when working with new technologies. It is an approach to learning new tools and technologies that we explored, but one that has a lot more possibility than that.

We should not forget that everyone can’t know everything in a project. It is preferable to create T-developers where everyone knows the basics of the projects, but different people specialize in different topics.

Relation to literature and guest lectures (how do your reflections relate to what others have to say?)

Literature: The Scrum Guide, written by Jeff Sutherland and Ken Schwaber https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf

Literature: Continuous Integration, Scaled Agile, Inc. https://www.scaledagileframework.com/continuous-integration/

A: For this project, we decided on having 2 product owners. While it is recommended to only have one, this wasn't a problem since our product owners wanted to share this responsibility.

We felt like the daily scrum played an important part for the team's ability to work efficiently. The standups let us chip out our work schedules for the coming days, all while knowing what the other team members were doing and when they would be available. The importance of the daily scrum meeting is confirmed by Sutherland and Schwaber in The Scrum Guide, where they stress the role it plays in improving communications and promoting quick decision making.

Sutherland and Schwaber also promote the use of sprint planning meetings, sprint reviews and retrospectives as tools to make the team more effective. We agree with them on these points as well as we felt like spending some time with reviewing the last sprint and its results, as well as planning what tasks should be completed in the next sprint, gave us a good idea of our current position and what we needed to do to take another step towards a functional product.

The product owner shouldn't be part of the development/delivery team, but due to the nature of this being a programming related course, our product owners naturally had to contribute to each sprint increment. We noticed the harm of this circumstance however. With everyone being a part of the development, the team lost a true outside perspective that would bridge the gap between the developers and the stakeholders.

The Scrum guide also stresses the importance of a definition of done that acts as a minimum requirement. While we had a definition of done, it should have been better defined in order to avoid ambiguity. Maybe would have been useful to lower its bar as well, to ensure that it could always be met without requiring a lot of work that may not be critical.

According to Scaled Agile, Inc., continuous integration is important for several reasons, one of them being the automated testing of new features. This was something that we implemented early on in this project, and it helped us make sure that the functionality we implemented was high quality and had all the elements needed to make life easier for the other developers on the team.
Scaled Agile, Inc. also promotes the use of Continuous Exploration as a means of checking whether or not the team stays on track with what the customers want and evaluates the value proposition of the product. This is something that we didn't do, but we all agree that this would

have been beneficial.

We initially picked two-week sprints since that was common online, but that didn't fit the short timeline of this course project. It simply wasn't often enough to get sufficient amount of reflection done.

B: In future projects, we would like to have one product owner that can make decisions regarding what features are the most valuable and should be prioritized. The product owner should be someone outside of the team, to give a true outside perspective of the product and what functionality will bring the most value at any given time.

Having an official scrum master who could promote and teach the concepts of the Scrum guide is something that would greatly benefit the team's use of Scrum. The guide is short enough for anyone to get around to read, its authors are reliable and it covers all you would typically want.

We also want to work with continuous integration and exploration, to provide reality checks for the team and to design and implement new features along the way.

A → B: We see the motivation behind having only 1 product owner though and although we had 2 without falling into any traps, it may not go that smooth in future projects. Therefore, we will stick with 1 product owner next time.

We should also appoint a scrum master for the team, whose responsibility will be to promote and help the other team members to adopt the agile practices and get comfortable with the scrum way of working.