

**Method selection and planning**

**Group 17**

**Team Loading**

**Joel Warren  
Charlie Wilson  
Jake Keast  
Hari Bhandari  
Ishrit Thakur  
Joshua Hills**

## Method selection and planning 4. a)

### Previous team's approach

We decided that an agile approach to our software engineering would bring us many advantages, as it speeds up the game development and makes it well-organised and flexible. This made us approach the development by tackling many small projects spread across the team. However, we did not allocate specific team roles as needed for the traditional agile scrum. We did not want a designated product owner or scrum master as we were all involved with the development and organisation of the project. We also found that different subteams needed different structures, for instance, the team working on architecture found that having a flat team structure worked best as we all had the same level of knowledge and experience so sharing the team responsibilities made so much sense.

We mainly used a shared google drive to organise our files. We chose this for its familiarity and its compatibility with LucidChart, which we used to make our UML diagrams. Our first experience with UML was through a google doc extension, which we used to make our Gantt charts and project breakdown. These diagrams are very simple to make, but the UML language was not well known to us, and the extension was difficult to use. When we started to make architectural diagrams, we used LucidChart. It is a visually based tool helping us to make UML diagrams. We found that using LucidChart through the web browser is much easier. We also utilised the post-it note feature in LucidChart to keep track of things that needed to be done, if there were any questions we needed to ask other teams or if individual team members needed clarification on certain parts of the diagrams. This meant that we could work in our own time and also see what other people had been up to. When a task was completed, then the post-it note would be deleted and the discord would be messaged to let everyone know.

We found this kind of remote cloud based system very useful as it meant we could update diagrams remotely without needing a meeting as all team members could see any updates. It also meant that team members could work on the project whenever suited them without having to update other team members or call a meeting- as we are all students with different schedules, this worked perfectly for us.

GitHub allowed the implementation team to work locally on their devices and then collaborate with the other team members. It was more unfamiliar than other tools we used but proved to be one of the most useful. The team's website was hosted by GitHub, allowing it to be accessible by anyone with the URL.

LibGDX was a very new development tool for all of us however we picked it for a number of reasons. Firstly we found that it supported all the main OS as well as working in browsers. It also has a very active and big community with a discord where our programming team could ask for help and advice. It also has some very useful game examples, which helped our team to get started with how it all works. We considered other game engines like JMonkey and LITlengine, but we soon realised that LibGDX was considerably more well established than the others. This meant the community was more active and there were more resources available.

To organise our meetings and to talk about our project, we communicated over Discord. This app is familiar to all of us and lets us have chats between our smaller development teams. We set up several channels to organise our communication which meant the general team channel didn't get filled up with unnecessary messages and that sub-team-specific messages were kept to their respective channels. We also used Discord to host our meetings over the Christmas break, as the system allowed us to connect calendars so that we could all see when the meeting was going to be. It also meant that sub-teams could host their own meetings without needing to coordinate with the whole team.

#### Inheriting method selection and planning

The software engineering methods and collaboration tools described above were used by the previous team (Team 21). Team ownership of the project has since changed, with our team (Team 17) assuming responsibility.

Upon assuming responsibility for the project, for simplicity we also adopted all of the previous team's software engineering methods, development and collaboration tools.

- Collaboration tools used: Discord and Google Drive.
- Development tools used: LibGDX and GitHub.
- Modelling tools used: Lucidchart (also collaborative) and PlantUML.
- Software engineering methods: Agile development.

Our team had recently been working on a project where we used all of these collaboration, development and modelling tools (except Lucidchart). Since we had them setup already, it was easy to quickly convert our existing systems for this project. These tools are described in detail under the previous team's approach.

To model our gantt charts and updated project breakdown, PlantUML was used, a software used to create various UML diagrams using plain text language. This tool was suitable because of our team's prior experience with the tool, and support for agile development through easy adaptations to change within the project.

However, we had no prior experience with Lucidchart. The alternative we considered using was PlantUML (for architecture), due to our prior experience with this tool. However, since all the architecture was made in Lucidchart, we decided to take the time learning Lucidchart and then extending them from there, which saved having to completely redraw the architecture from scratch.

An agile approach to software development was also used primarily because of our team's prior experience with it, and its fitness for the project which is described in detail under the previous team's approach. Agile development also helped us to react quickly to unexpected requirements which were given by our customer.

#### **Method selection and planning 4. b)**

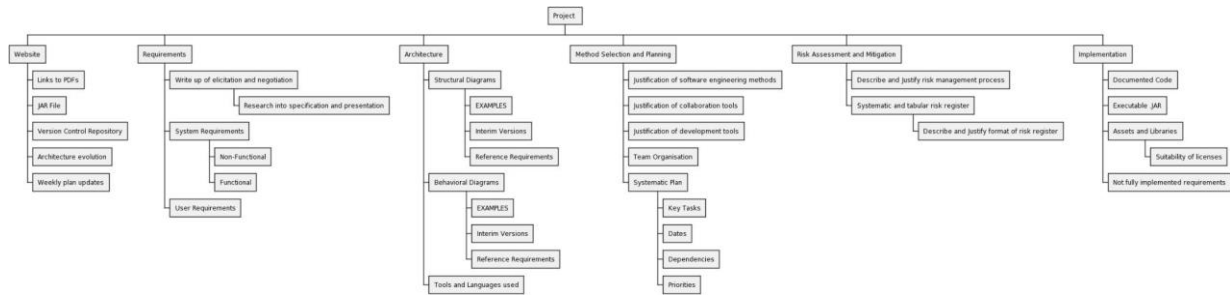


Fig.1

At the beginning of this project, we discussed our strengths and weaknesses regarding the skills needed to successfully make and document the game. To decide on our team's architecture, we wrote down all the deliverables [fig.1]. This helped us visualise the workload so we could split it fairly. It also helped us to follow an agile approach to this project as it meant we could focus on well organised smaller sub-problems. Individually we put ourselves forward for these tasks and explained why we were suited to them. Then as a team we made sure that every member was allocated to roughly 15 marks of the work, and that everyone would have work to do throughout the 11 weeks. At this point we decided that having three people coding was better than everyone, as it would ease communication and technical difficulties regarding pulling and pushing to the repository. Oscar, Immanuel, and Andrew were to implement the game. Scarlet, Andrew, and Madeleine focused on architecture. Josh and Immanuel worked on the risk register and mitigation. Madeleine made the website. Scarlet and Madeleine worked on the method selection and planning. Josh and Oscar both worked on the requirements.

We stuck to these roles well, but we made sure that we were lenient when needed. If a team needed support or a task was deemed more time-consuming than assumed, we set more team members to work on it. This happened mostly with the implementation, as it took more effort and time than expected. Due to this, Immanuel worked less on the risk assessment and Oscar worked less on the requirements. Even though this meant the marks were unevenly distributed, we decided that it was still the best course of action as they both had worked on their respective tasks enough nearer the beginning of the project.

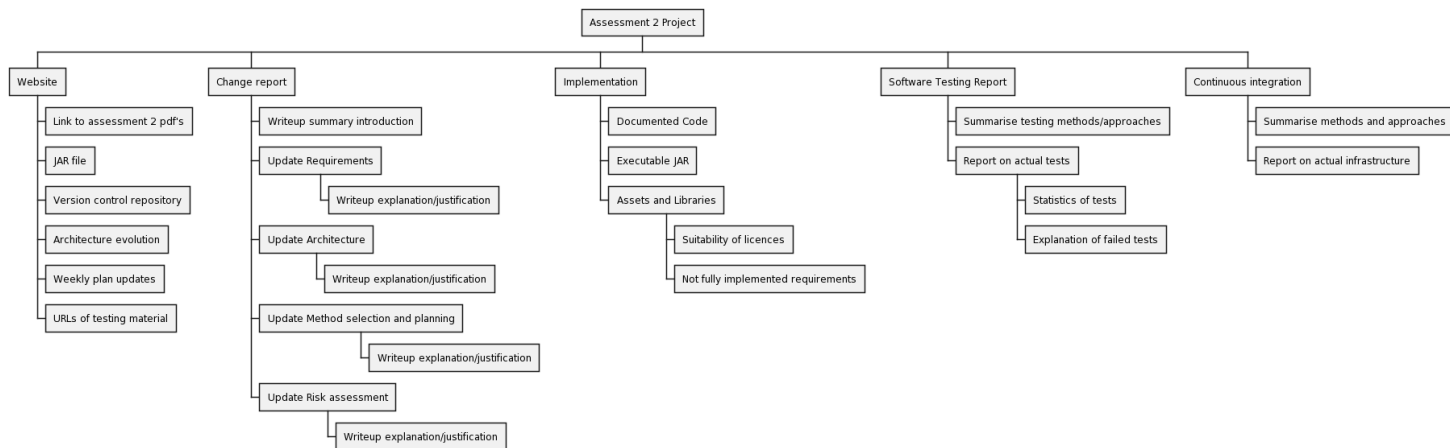


Fig. 4 - updated project breakdown

Team organisation has since adapted to reflect a change in team ownership of the project, as well as the evolution of the project.

To organise our team, we followed a similar approach by the previous team by discussing our strengths and weaknesses early on to determine best fits for deliverable ownerships. This was mostly already predetermined from a previous project, whereby team members continued working on areas they were confident in. We also took into consideration that members should be roughly allocated 15 marks each. The various deliverables ownership are listed below:

- Website - Hari, Charlie
- Change report - Joel, Ishrit and Joshua
- Implementation
  - Coding - Charlie, Jake
  - Licencing writeup - Joel (Charlie as backup)
- Software Testing Report - Hari (assisted by everyone)
- Continuous Integration Report - Joel, Joshua, Ishrit

Likewise, this was adapted when necessary. Initially, Hari was assigned to work on the change report. However, after further consideration we decided that this deliverable did not need 4 team members, hence Hari began working on software testing instead which was determined to be a higher priority deliverable.

This approach to team organisation was well suited for the team because everyone was working to their strengths and suitable for the project because each deliverable had at least 2 members, thus ensuring a low bus factor to avoid risk (R1.01). The adaptive approach also suited agile development.

However, after further consideration, it would have likely been better to have all team members contribute a small part to the implementation, to gain a better understanding of the code. Towards the end of the project, it was then impractical to assign other team members

to help Charlie and Jake with implementation, being so late into the project it was then difficult to understand the code and to actually assist with coding.

## Method selection and planning 4. c)

### Previous team's systematic plan

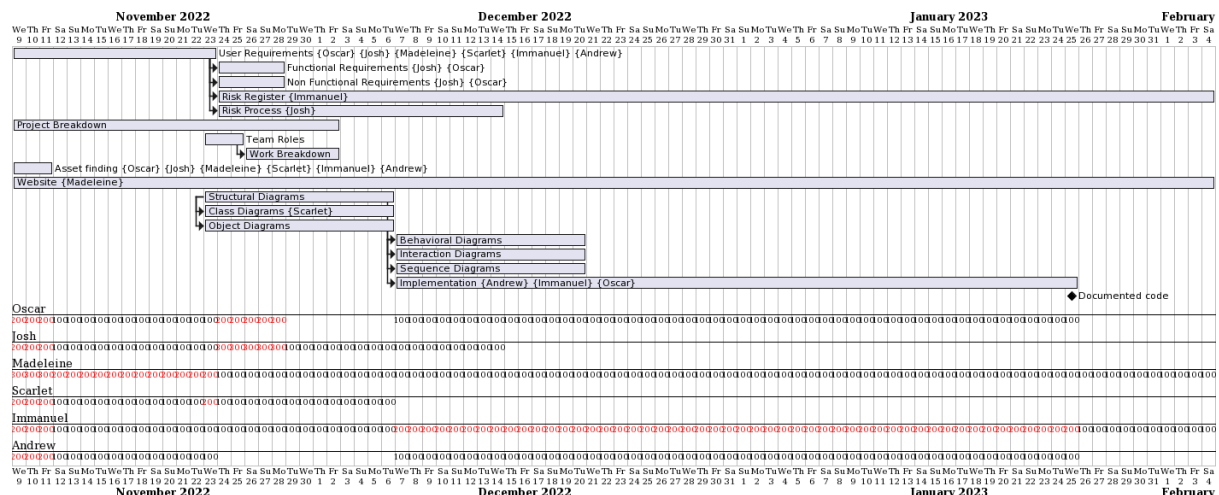


fig.2

After we decided on our team's organisation we made a gantt chart [fig.2]. This evolved very quickly as we soon realised that we were too optimistic with our time frames and our schedule was unrealistic. At the beginning of the project we were too focused on the requirements and risk compared to the architecture and implementation. We also were too shortsighted with our plan and didn't consider alternatives as seen in our very first gantt chart [[imozwastaken.github.io/snapshots.html#gantt](https://imozwastaken.github.io/snapshots.html#gantt)].

There are many differences between our initial plans [[imozwastaken.github.io/snapshots.html](https://imozwastaken.github.io/snapshots.html)] and our final plan [fig.3]. We extended all the write-ups till the end of the project as there was always new content to be added as we changed our work. We also made architecture more of a priority. We managed this by splitting it into smaller tasks and extending the time we were working on them for. We also decided that we would make state and class diagrams, and CRC cards, but no interaction or sequence diagrams. In fig.1 there are no CRC cards, and there is an obsolete 'object diagram' task. The class diagrams should also be dependent on or parallel to the CRC cards.

In fig.1 it shows that the implementation was to start at the end of November. However, this is incorrect and a simple typing error in UML; implementation started properly around a week before this.

It was made clear around Christmas that we needed to factor in more time to allow the implementation team to familiarise themselves with the game engine. This subsequently pushed back the coding schedule. Only once the coding had fully begun had we thought of

all the sub tasks needed. The implementation tasks had the most dependencies compared to the other deliverables, as there has to be basic functionalities before design work. After the map was completed and assets were put into the game, the implementation developed very quickly. In the last two weeks of the project our plan did not change. There was editing needed and formatting, but nothing new started.

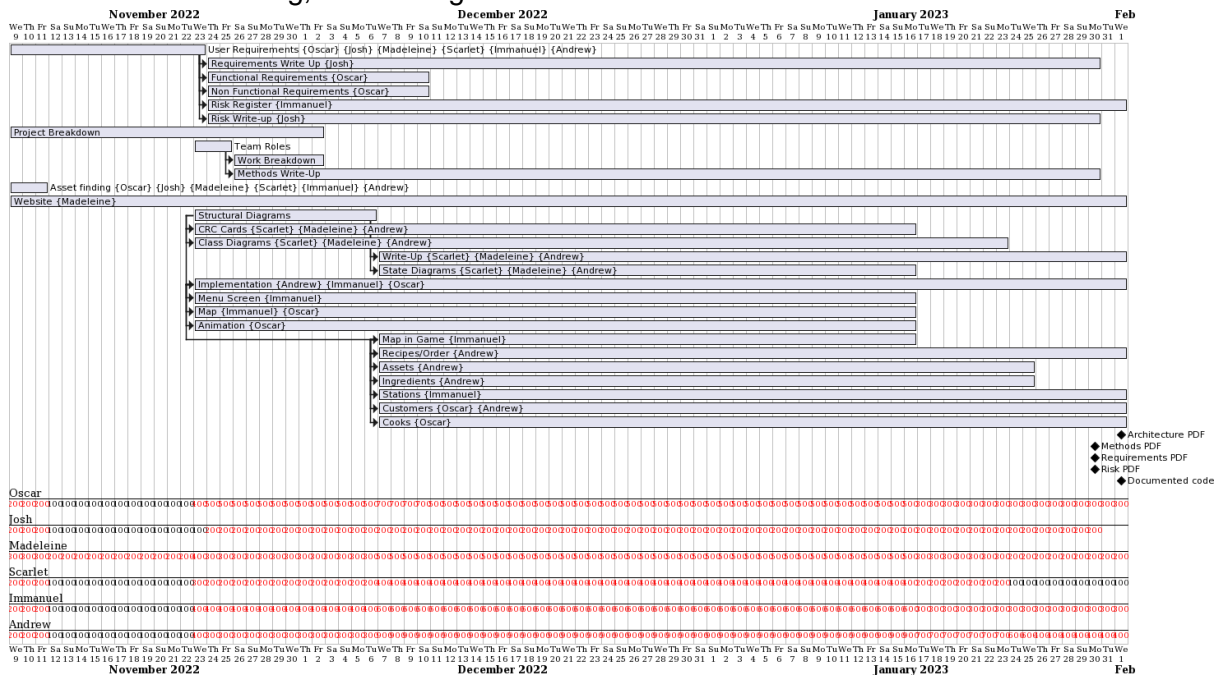


fig.3

One of the problems we had with such a flexible project and team architecture is that it was too open to change so often the design team couldn't keep up with the changes being made by the programming team. This could have been avoided in a number of ways, firstly having an assigned project manager, who was also part of one of the design teams and the programming team would have kept everyone on the same page. However, that might not have been possible because that would be a lot of responsibility and work for one person and would have meant that other team members might not pull their weight as much as they should. A different way to solve this problem would be to have one person, not the project manager, who is in both teams to keep everyone on track.

The final solution would be to select a more appropriate architecture for the programming. I think if we had used the pipeline architecture, then that would have allowed a little more flexibility and would have meant that we wouldn't have had to complete all the design work before beginning coding, it could all be done one after the other, because whilst the programming team were working on the first section the design team could work on the second section. This would probably have required more meetings as the design team wouldn't have been able to see all the issues the coding might bump into, but it may have made both jobs a little easier as there would have been fewer changes.

Updated systematic plan

The systematic plan described above was carried out by the previous team (Team 21). Team ownership of the project has since changed, with our team (Team 17) assuming responsibility. Furthermore, the project has since evolved, with new deliverables required. Hence, the systematic plan needed to be updated.

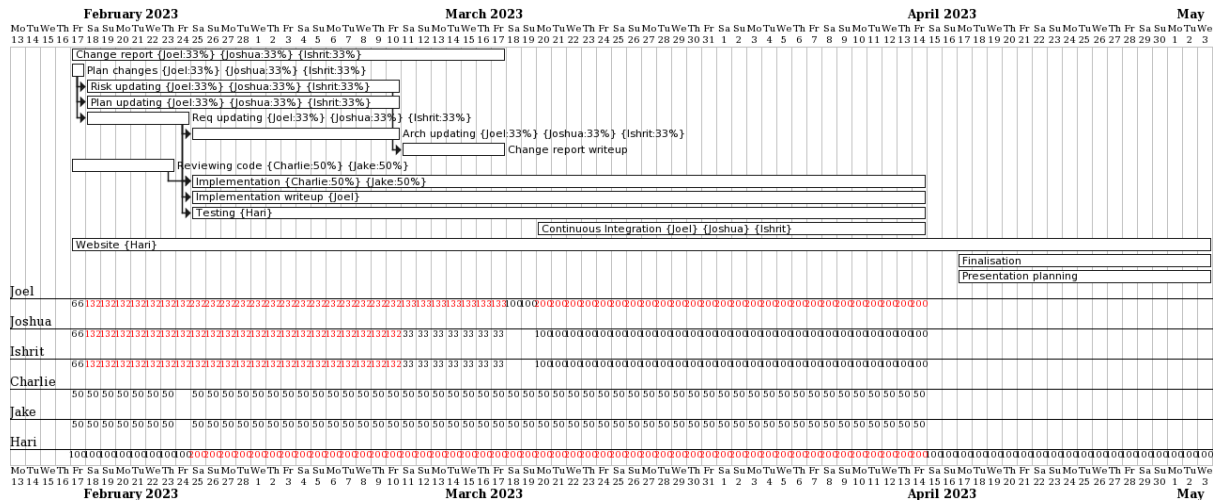


Fig 5 (new initial gantt chart)

As described above, a new gantt chart needed to be created when our team took responsibility for the project, completely changing the plan. This was driven by a change in deliverables and requirements given by assessment 2, thus requiring a new gantt chart to represent the updated project breakdown (fig 4). This is similar to the previous teams gantt chart in format etc, with just different deliverables and different time frames, as well as different team members.

The gantt chart was updated weekly, with snapshots available at:  
[https://eng1-loading.github.io/assessment\\_2-website/snapshots2.html](https://eng1-loading.github.io/assessment_2-website/snapshots2.html)

The final gantt chart is shown below:

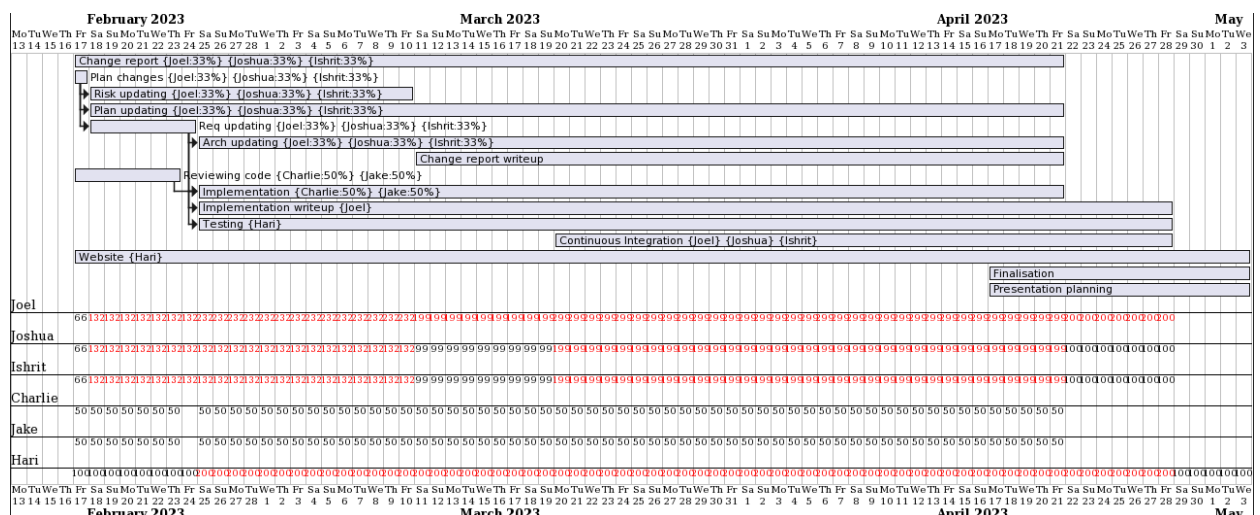


Fig 6 - final gantt chart.



The final gantt chart (fig 6) shows how the plan has further evolved since our team took responsibility for the project. Initially, the plan was to have all deliverables/implementation finished by Friday 17th April, to be mostly complete by the end of the easter holidays and only have finalisation to do. However, in reality this didn't happen, as is shown by the final gantt chart. Certain deliverables, such as updating architecture took a lot longer than expected, causing the deadlines for architecture updating to be extended. This also caused the change report writeup to be delayed, which then had to have its deadline extended.

Work productivity over the easter holidays (Friday 17th March - Friday 14 April) was quite low. Continuous integration was not started and implementation/testing was not finished by the initial deadlines. Therefore, again these deadlines had to be extended.

Apart from deadlines for deliverables being extended, the gantt chart from fig 5 to fig 6 changed very little in terms of structure/member assignment etc.

Upon further reflection, repeatedly extending the deadlines of incomplete deliverables may not have been the optimal solution. Instead, we should have reprioritised work where needed and assign more team members where possible, such as to the architecture. That way, any dependencies (such as the change report) therefore won't be held up.