Method Selection & Planning

Group 18

Team B

Olivia Betts
Zac Bhumgara
Nursyarmila Ahmad Shukri
Cameron Duncan-Johal
Muaz Waqas
Oliver Northwood
Teddy Seddon

Software Engineering Methods

Our team uses **Agile software development** as our software engineering method:

- Short development cycles are used in the agile project management technique, which prioritises continuous improvement in the creation of products and services.
- Requirements and solutions are developed in collaboration amongst self-organising cross-functional teams.
- We chose this to ensure we were constantly checking the code we had produced against the requirements.
- Other development methodologies considered but ultimately discarded were the waterfall method (outdated and not useful for changing requirements) and spiral (ideal for large, risky projects - neither of which this is).

For our game engine we chose to use **LibGDX**:

- This was the game engine which after our research, we found to be easiest to integrate along with third party libraries.
- The relevant third party libraries all had thorough documentation available online too, thereby helping to mitigate the R_LIBRARIES risk.
- In general the majority of the LibGDX framework is written in Java, which our whole team is skilled at.
- We considered LitiEngine, but decided against it, as the guidance available online was more thorough for LibGDX.

For our IDE, we used **IntelliJ Idea**:

- IntelliJ IDEA is one of the recommended IDEs from the LibGDX site. There was a lot of functional support available from LibGDX in general.
- IntelliJ can also integrate a project from Github (our chosen repo), so it would be very simple to update our existing codes and commit and push any changes.
- IntelliJ also has colour coded specific data types and comments and can generate any methods, constructors and javadocs automatically. This helped to mitigate the R_FUTURE_PROOF risk; commenting the code was simpler.
- We considered using Visual Studio Code, but as IntelliJ was developed specifically for Java, it proved to have more machine intelligence.
- The ‘refactor’ element in IntelliJ was particularly useful as it allowed us to change the names of packages and classes and any places where that name would change automatically.

For our Version Control System, we used **Github**:

- This helped to avoid code collisions and made sure that they do not adversely affect the main branch of the code. This helped to mitigate the R_COMMIT risk.
- Github, as a form of Cloud storage, was used to store our code files. This made collaboration on our project easier.
- Multiple people could work on the same codebase simultaneously, which again encouraged remote cooperation.
- As mentioned before, Github can integrate into IntelliJ, making it very easy to push and pull changes.

For communication, we used **Discord** and **Gmail**:

- We chose Discord because it proved to be the form of communication which everyone was familiar with.
- Discord allows separate channels for different topics - subsequently we created different channels for each of the deliverables.
- Discord offers a voice channel in which we would occasionally host online meetings, especially during times when people were not at university, over the holidays.
- Another useful feature is the ‘share screen’ feature as it allows better collaboration during any online meetings.
- Our group email would send an email message to everyone at once so everyone receives any new updates through email. In this way we were also able to address the group as a whole
- Having two separate methods of communication helped mitigate the R_COMMS risk, as if someone was not replying on Discord then we could contact them by email and vice versa.

For storage and collaboration of our deliverables, we used **Google Docs**:

- The version control functionality allows multiple people to edit the same document.
- The documentation is all stored in a shared Google Drive, so they can be accessed easily and from multiple devices.
- As a form of Cloud storage, these were accessible from anywhere and were a backup to the local copies saved on our devices. They could be recovered from here in case a file is accidentally deleted. This helped to mitigate the R_DELETE risk.
- Another bonus of using Google Docs was that the files can easily be downloaded to .pdf format, which was the required format for the deliverables.
- We also used **Google Sheets** to create the Gantt chart for our project. This was easiest to implement and also easy to alter and clone for any further Gantt charts.
- We considered using OneDrive, however it took longer to set-up and more difficult to ensure everyone had access, therefore opted for Google Docs instead.
- We also considered using Microsoft Word, but again opted for the collaborative element over the slightly extra functionality.

To create the behavioural and structural diagrams for the UML for our architecture, we used **PlantUML**:

- This proved to be one of the easiest tools to produce accurate UML diagrams, with extensive guidance online and in our Q&As.
- We considered LucidChart due to its link to Google Drive, but decided against it, due to PlantUML’s ease of use and available guidance.

Team Organisation

Our team split into smaller sub-groups, each of which focused on one section of the deliverables. We then had one spokesperson (Zac Bhumgara) who coordinated communication with the customer, for example, emailing to check about the requirements, as well as between sub-groups. It was important to have one main point of contact, as otherwise multiple people might have been sending the same messages separately to the customer.

In these sub-groups, each organised themselves, whether that was working in a pair, or working in up to a group of 4. This was ideal, as it meant a sub-group could meet and discuss their specific problems without having to coordinate the entire group to do so. This approach worked for the project as a team of 7 people would have been too large for everyone to try and know everything happening at once. It would also have been unnecessary - for example, the method selection and planning requirement was too small for 7 people to work on at once.
The majority of sub-groups had 2 or more people. This meant that if one person was unable to complete their portion of the work - for whatever reason - there was at least one other person able to step in and do it. The exception to this was the website, which was worked on by just Olivia Betts. However, in this case the site code was also entirely on Github and documented, so another person could easily take up the work if they couldn't complete it.

We organised ourselves as follows:

Website	Olivia Betts
Requirements	Olivia Betts, Zac Bhumgara, Nursyarmila Ahmad Shukri
Architecture	Oliver Northwood, Nursyarmila Ahmad Shukri, Muaz Waqas, Teddy Seddon
Method selection and planning	Nursyarmila Ahmad Shukri, Cameron Duncan-Johal
Risk assessment and mitigation	Olivia Betts, Cameron Duncan-Johal
Implementation	Muaz Waqas, Oliver Northwood, Zac Bhumgara, Teddy Seddon

This meant we could ensure everyone got a fair share of the work, so nobody was given too little or too much (a number that worked out to ~12.8 marks of work per person)

Key Tasks

Task	Start date	End date	Dependencies
Requirements Elicitation	16/11/22	30/11/22	N/A
Initial Architecture	23/11/22	30/11/22	Requirements elicited (doesn't have to be perfectly finalised)
Risk Assessment	7/12/22	14/12/22	N/A
Implementation	18/1/23	31/1/23	Requirements, Architecture
Architecture evaluation	25/1/23	31/1/23	Implementation, Initial architecture

(We have also made a Gantt chart laying this out visually, which can be found on our website TeamBEng1.github.io)

The above chart uses very broad definitions of key tasks. We found this suited our way of working best, with individual smaller teams being able to dedicate the time they saw fit to complete the work in ways that worked for them, whilst remaining within the overall deadline.

As an example, the Risk Assessment was further split into two key tasks:

Task	Start date	End date	Dependencies
Identify risks	7/12/22	8/12/22	N/A
Risk writeup	9/12/22	14/12/22	Identify risks

Splitting of tasks:

Similarly, the requirements were also split up into key tasks. This included the customer meeting, from which the requirements are elicited. Each UR, FR and NFR is then outlined (and given a specific ID). At the end of the requirements section, the final key task would be to consider use cases, and see if the requirements match up in each instance.

For architecture, we planned to create an initial architecture design at the start. This would depend on the requirements (as the classes we would design would be based on what the customer is asking for). So at the start, we created some diagrams for how the game should look like and following from this we made some CRC cards, showing the different classes, responsibilities and collaborations in the game. Alongside the implementation, we then began creating the final version of the architecture. As expected, there were major changes from our initial design; these changes were identified and the transitions were justified. At the end of our implementation, we added some of the classes into the architecture which we were not able to implement but were part of the requirements.

For methodology and planning, at the start of the project we selected our chosen software engineering method and planned out the whole project by developing a Gantt chart. Further on from that, we would create a weekly ‘to-do list’ so that everyone in the group had a goal to work towards before our next meeting. This would ensure that no-one remains idle, because they don’t know what else they can do. Further Gantt charts were made weekly to remind ourselves of where we should be, and in case we had to catch up. These can be found on the website, for which the link is posted above.

For implementation, planning was especially important. Implementation was dependent on the initial architecture, so it was of utmost importance that the initial architecture design was complete before the intended start date of the implementation. Alongside coding the game, it was important to document at the same time, using Javadocs and comments. This would ensure that other members of the implementation team would be able to understand what the code is trying to achieve.
During the implementation, before starting to use a specific library, we researched its licensing online. If the licensing allowed us to use it, we added it to our project and made a note of it in the implementation document. Moreover, at the end of the implementation we looked back upon the requirements and identified which of them we did not/were unable to implement. Again we noted this down in our implementation documentation.

Importance of tasks:

We dedicated a priority of tasks to each task. Usually, any tasks which dictate a further task were regarded as high priority. We would ensure that these were completed before their deadlines, so that it does not lead to a ‘lag’ towards starting the next task.

In our project, we identified the requirements elicitation as being high priority (as the initial architecture depended on it). The initial architecture itself was also high priority, as it dictated how we would start to actually implement the game. The implementation itself was recognised as important as it involved making the product which the customer expects by the end of the deadline. For this reason, it was given the longest share of time.

Critical Path:

The critical path is a modelling method which determines the optimal and chronological timeline to complete your project. In this case, we identified the critical path to be **Requirements Elicitation → Risk Assessment → Initial Architecture Design → Implementation → Review → Final Architecture Design**

Bibliography:

https://www.wrike.com/project-management-guide/faq/what-is-critical-path-in-project-management/