Change Report

Galin Dzhumakov

Aisyah Firoz Khan

Faran Lane

Samuel Nicholson

Jack Polson

Alana Witten

Part 2a) :

We made sure to get all the other teams' artefacts in an editable form. Initially, we had a meeting to discuss what changes we wanted to make to each deliverable. In this meeting we used the convention of turning text red if we wanted to remove it and adding new text in blue. This allowed us to easily track what changes we have made. We also kept three versions of each deliverable: an old version, a new version and an intermediate version that kept the different coloured text. We asked the other group for changes that the client suggested so we could add these to any changes we thought to make. We chose Google Docs to track our changes as it allows multiple people to work on the document simultaneously, make comments to allow for reviewing of changes and the ability to suggest changes directly. We also made sure to get their source code and their architecture in PlantUML form so we could make direct changes to their work.

Against our new requirements, we created a to-do list of what needed to be changed as well as a priority of what shall and should be implemented. We regularly committed our code to track individual changes that have been made.

Part 2b)i : Requirements

We made very minor changes to their introduction of how requirements were elicited and presented. The original can be found here: https://eng-25.github.io/Req1.pdf.This is because we followed a very similar process. The only difference is that we did not make contact with The University of York Communications Office so we removed this point as seen in the updated version here: https://eng-25.github.io/Req2.pdf. Although we followed a similar format for how our requirements were displayed, we did not follow their Natural Language format. We also included a brief point about how we elicited requirements and updated the table after receiving an updated brief.

We made minor changes to the language included in their table due to the change in requirements. An example of this is for the requirement UR_CONTROL_COOKS. In the original version it states there are two cooks whereas we changed it to just describe the process of moving them. As there is now a different number of cooks depending on the mode/progress, this better tracks to the requirements. We split certain user requirements into functional requirements due to their being too much detail that is no longer applicable to the new requirements. UR_COOK_FOOD originally stated the recipes but as there is now a difference in recipes depending on the mode chosen, this is a more accurate description. Due to updates in the requirements, we also updated certain requirements. UR_CUSTOMERS originally said that each customer requires one dish but the new brief states that a customer can ask for more than one dish so this needed to be updated. We removed certain requirements that we deemed unnecessary (e.g. UR_BRANDING). We noticed that there were requirements in the brief from assessment 1 that were missing from their table. One example of this was a tracker for the number of reputation points. Therefore, we added this requirement. We added new functional requirements that specifies the distinction between the two modes. This is included in the ID as the suffixes SCENARIO and ENDLESS. We fixed an inconsistency in their table also. They had a functional requirement FR_CHANGE_PLAYABLE_CHARACTER which linked to a user requirement UR_CONTROL_CHEFS. This user requirement does not exist so as this was an issue of updating the original name and not its reference, we changed it.

Part 2b)ii : Architecture

After eliciting initial requirements, we first decided to update the old class diagrams. In our first iteration of the main class diagram, we knew the game would now have to have 2 modes - scenario and endless, so we changed the GameScreen class to be abstract, with 2 child classes for the 2 different modes. From analysing the code given to us, we could determine the CustomerManager could only handle 1 order (Recipe) at a time, and that each order was made up of exactly 1 Recipe, so we decided to add an intermediate Customer class between CustomerManager and Recipe classes. This meant the CustomerManager could still use a lot of its previous logic written to handle a Recipe, instead handling Customers. The customer class would then handle a List<Recipe> to allow for different sized orders. We added a new OvenStation class, which we planned to use for the new recipes in the game - pizza and jacket potato. This would allow us to add some new, more challenging game mechanics, making endless mode more complex. We also added the PowerupManager class, following the previous code's design to use managers linking to the GameScreen, each manager handling multiple instances of its managed class. We made the PowerupManager handle multiple instances of the abstract Powerup class, as we didn't know which powerups we would add and therefore couldn't determine exactly how we would implement their specific classes. Finally, we packaged up the classes based on the code given to us and added some missing classes we felt were important to the diagram, such as all the Ingredient and Recipe child classes.

After some time implementing, and upon further reflection of the previous code given to us, we created a second iteration of the main class diagram. We had realised the GameScreen class itself handled very little to no actual game logic and instead most classes were added to a main libgdx Stage, which would call an act() method on all of its children. This meant having 2 new GameScreen instances would be rather obsolete and a lot of unnecessary work. Instead, we decided to add an isScenario field to the GameScreen class which would be passed into any classes which needed to handle differing logic based on game modes, allowing them to maintain the design of handling their own logic when act() is called. Secondly, we made OvenStation now extend CookingStation. We wanted the OvenStation to act similarly to the CookingStation but handle different types of inputs, so it made sense to have this relation. On top of this, we added the IFailable interface for the CookingStation and OvenStation classes to use, as they would be the stations with failable steps. We had now begun to implement specific powerups, so we could refine our Powerup classes. TimedPowerup would act as a base for any lasting powerup over a period of time, whilst the ISingleUsePowerup could be implemented in any single use powerups, allowing them to have their own unique structures and logic but ensuring they implemented an activate method. Finally, we reduced their child Ingredient and Recipe classes as we had realised they were not needed at all and would in most cases just call super constructor and implement no extra functionality.

In our new game class diagram, (link to their old one here) we added new GameScreen fields to account for difficulty and multiple game modes. In the Chef class, we added a speedMultiplier field, which would be used for an easily modifiable movement speed (useful for a powerup for example). We added a Customer list to CustomerManager, as well as the

getNewCustomer() method to represent the fact that the CustomerManager now handled Customers not Recipes and the randomPowerup() method to show that powerup events happen in the CustomerManager – in our implementation, powerups would have a chance to randomly trigger upon serving a customer. In the Station class, we added a clearStation() method, a new functionality which we felt was essential for smoother gameplay, allowing players to clear a station holding ingredients at any point. We also added locked and price fields, as well as a buy method to show that stations could be locked and bought. Finally, we added relevant loading and saving methods as this was a big new functionality in the software.

We did not change any logic or code relevant to the Observer and Subject interfaces, so their class and sequence diagrams did not need any changes.

In our new stations class diagram, (link to their old here) we added the new OvenStation class. As it was a subclass of CookingStation, we also had to change some methods in CookingStation to be protected, not private. Finally, we had changed RecipeStation to using a Map instead of integer fields to represent held ingredient counts. This made it easier to store and handle in one variable, as well as making it much more scalable. A similar map was also added to OvenStation.

Lastly, we changed the old UI class diagram. In our new diagram, we added relevant failbar render methods to StationActionUI and StationUIController classes, wanting to replicate the progress bar methods already existing. The FinishGameUI method now had passed in parameters as it would hand off data to the game's EndOverlay, as opposed to simply displaying text on the game UI when the game was finished. Importantly, we added resize methods. The previous code had not implemented much resizing logic and had only used some percent values in some places. Implementing a resize method would allow us to make sure all the UI would resize properly. We also added update methods for new UI elements, money and reputation. Finally, we added the addRecipeGroup() method, as orders were now groups of recipes, it allowed us to implement a system to render dynamically sized and changing orders.

As well as class diagrams, we had to make changes to the previous behavioural diagrams given to us. Firstly, we changed the old ChefSelected state diagram (link to our new one here) to account for the possibility of more than 2 chefs now being possible.

As previously mentioned, the collider sequence diagram did not need any changes. The chef state diagram also did not need any changes, as the logic was still the same.

We updated the old Screens state diagram (link to our new one here) to account for the new difficulty and load screens, as well as an exit button. We also made changes to represent pausing and ending the game, via both save and exit as well as the game finishing.

Part 2b)iii: Method selection and planning

Upon inspection of the method selection and planning document we inherited, we did not feel that any change to part A was necessary. This was due to the fact that the software engineering methods and the development or collaboration tools that the previous team had used to support their project were the same as the ones we have been utilising.

When we were planning our project we chose to use GitHub to host and share our code as it allowed us to collaborate with each other on the project. Our IDE of choice was IntelliJ due to its advanced debugging features and it is available on multiple operating systems. Furthermore, our chosen method of communication was Discord as all members were familiar with it and used it regularly. Finally, we used Google Drive to share and collaborate on the documentation associated with the project. These pieces of software are the same as those used by the team of developers we inherited from and therefore no change is required for this part of the document.

Secondly, moving onto part B of the deliverable, we had a very similar approach to the previous team's organisation and the planning approach to the project. We also assigned the roles of Meeting Chair, Secretary, Librarian, and Report Editor to different team members; however, we also employed a shadow system that the previous team had not incorporated. This change can be seen within the new version of the Method selection and planning document here https://eng-25.github.io/Plan2.pdf. The table (Figure 1) now shows who within our team was responsible for each role as well as who was assigned to shadow each role. The previous teams table can be found within the document https://eng-25.github.io/Plan1.pdf. We felt this was a necessary change to make to the document as we found the shadowing system we had in place to be extremely useful and we felt that we would continue to obtain value from it with it incorporated into the planning of the next stage of the project.

Finally, moving onto part C of the deliverable which outlines the previous team's systematic plan for their project. We have decided to leave their plan in place untouched, we have done this so that we can show the overall plan of the whole project. Therefore, the changes we have made to this part of the deliverable is the addition of our systematic plan for the project since inheriting it until completion. This can be found on pages 7 and 8 of the new deliverable: https://eng-25.github.io/Plan2.pdf. We have made these changes to the document so that it provides us with a clear roadmap for the project, showing the ideal timelines for different tasks and dependencies between tasks. It is also important to have this plan in place to improve our time management and such tasks are completed when they need to be. We felt that leaving the previous team's plan untouched and just adding ours beneath was the best way to change this part of the document as it will show to our stakeholders the complete journey of the product and how it has been produced from start to finish.

Part 2b)iv : Risk assessment and mitigation

In order for us to make progress with our game and continue development from the project we inherited, it was an important task to analyse and update the risk registers we inherited. In order for us to continue to monitor all possible risks, the likelihood and severities, we had to make changes to this document. The risk deliverable prior to any changes made by us can be seen here, https://eng-25.github.io/Risk1.pdf.

We started by going through the risks that the previous developers had identified and seeing which of these did and did not apply to us as well as the second half of the project rather than the first part. We deemed that all of the risks identified by the previous developers were still applicable to us as we moved forward into the next stage of updating the risk register.

As a team we discussed potential risks not included by the original developers. Our finalised risk register, including our new added risks, can be seen here https://eng-25.github.io/Risk2.pdf. Firstly, we tried to identify any risks associated with the project that we deemed were a risk but had yet not been included, these are risks R_PROJECT_07 and R_PROJECT_09. Furthermore we then identified risks associated with the product that we felt had been missed when the previous engineers had drafted their risks. These are referenced in our new Product risk table as, R_PRODUCT_04 and R_PRODUCT_05. We added these risks because we felt that it was a possibility that these could take place and therefore a mitigation and an owner should be in order to reduce the resulting effect if these did happen.

Furthermore, as a team we had a meeting to discuss and pinpoint all of the potential risks of inheriting a project from another team. We felt this was an important addition to our risk documentation and necessary to be included as we felt there were several things that could potentially go wrong with inheriting another team's code and documentation. Initially we were concerned of the risk of not being able to contact the previous developers or an inability to access their documentation to aid our further development, so we came up with mitigations for these potential risks. These risks are referred to as R_PROJECT_08 and R_PROJECT_10 respectively within the new Project risks table.

Progressing on, we moved on to adding the risks associated with inheriting another team's code. These risks included, misunderstanding code/poorly written code, bugs within code and game features being incomplete or missing. These are referenced as R_PROJECT_05, R_PROJECT_07 and R_PROJECT_08 respectively. We felt that updating the risk registers with the risks associated with continuing someone else's code was a necessary change to the document. This is because it allowed us to better understand the potential challenges and plan accordingly, this also helped us to minimise the impact of these risks on the project and ensure our new code is well-maintained and meets the project requirements.

One other change we have made to the risk documentation is how the severity and likeness of a risk is measured. The group we inherited the document from used a rating system of 1-5 with 1 being low severity/likeness and 5 being high. However we felt this was necessary to

change to a rating of L/M/H (Low/Medium/High). We made this change to allow us to effectively colour code our risk register as we feel this visual aid makes each risk easier to read and understand, allowing us to quickly identify the most critical risks. Furthermore, this addition helps to keep consistency within documentation so that everyone on the team understands and interprets the risk register in the same way.

Finally the only thing we had left to change within our newly developed risk document was the designated owners of all risks. This was an essential change as we needed to ensure that risks were being actively monitored and managed and that there is clear communication and accountability for risk-related decisions.