Change Report

Team 7 - Broken Designers

Adam Brown
Morgan Francis
Shabari Jagadeeswaran
Oliver Johnstone
Laura Mata Le Bot
Rebecca Stone

Introduction

After inheriting the project from the other team, we held a group meeting to decide on a plan and split the work between team members. We approached this task by listening in turn to each team member's preferred tasks for assessment 2. We split the change report between 5 of us, each choosing a different section to work on. We ensured that the section each member was working on was different to that of assessment 1, with the member responsible for that section in the previous assessment acting as their shadow. This was agreed upon during our Sailboat retrospective, as mentioned in the method selection and planning report for this assessment. The role of a shadow was to overlook a section of the report, helping out where needed and making sure work was getting done in a timely manner.

As we had already looked through our chosen group's deliverables when choosing a project to take over, we had a good idea of what needed to be edited and updated when taking over their project. We updated the requirements at the beginning of the project, amending it according to the new assessment brief. We also considered the elements of the other team's risk register, adding and deleting where necessary for our position at the start of the project. We constructed a new architecture, based upon the previous team's diagram, but including the new requirements and allowing for testing to take place.

One of the most helpful tools on the documentation side of the project was the feedback given for the previous assessment. We studied our feedback and considered that of the other team, organising the key points into a feedback analysis table. This considered the positive aspects of our work and any improvements that could be made. Summarising and comparing both our feedback and that of the other team in a conclusions column allowed us to improve upon their deliverables effectively. It also gave us a good idea of what needed to be done for each deliverable before we started work on them.

Generally, when working on the deliverables, we began by editing the other team's submitted deliverables and keeping a note of any changes we made in a change report. This focussed mainly on making it more readable and conveying the points we wished to make effectively. We continually added to the change report throughout the project if any problems arose that required us to change it. Next, we focussed on any additions that we wanted to make to the deliverable as a result of a difference in the two teams' approaches to the project, especially affecting method selection and planning and the risk register. New requirements provided in assessment two had to be added to the requirements and architecture deliverables.

In terms of implementation, our main focus for the first couple of weeks was refactoring the inherited code and setting up the testing environment to allow us to begin both testing and implementation synchronously using test-first development. We found it useful to have an in person meeting with limited members of the team who were tasked with the testing or implementation aspects of the project. This ensured testers all knew how the environment worked and were on the same page when they later began writing tests. Those working on refactoring the code could split the task between themselves while not using any of the others' time as a full team meeting would have.

Requirements

Original: Req1.pdf

The requirements report we took on was very similar to the one we had made for our previous project. However, we decided as a group that there were changes we had to make in order to improve its quality as well as to make it fit the new requirements we have been given.

The first thing we changed was the formatting of the previous teams requirements table. The other team never mentioned using a formatting style, and the table didn't seem to follow one either. From previous research, we decided that refactoring the previous teams table to follow the IEEE Standard Requirements Engineering document [1] would give us the best result. This document suggests creating generalised and brief user requirements, which are then referred to by more detailed functional and non-functional requirements. Our decision to change the requirements table in this table in this manner was driven by research into requirements documentation. IEEE's [1] methodology allows for easy to read and understand formatting. Additionally, it suggests grouping requirements together which helps when delegating tasks amongst team members.

After deciding to follow the IEEE Standard Requirements Engineering document [1], we refactored the requirements table around it. This meant removing, replacing and adding to the existing table. Examples of this include the refactoring of the UR_DISH_SERVE, UR_WIN and many more user requirements into functional requirements. Under the new formatting style, many of the previous teams requirements were too specific to be user requirements. This was fixed by making them functional requirements and introducing newer and broader user requirements that they could link to.

In addition to formatting changes, we also added to the requirements table in order to include additions that covered the new design requirements. New user requirements such as UR_SAVE and UR_MODE were added to broadly cover the new save feature and difficulty feature that was requested by the customer. These requirements were further elaborated by new functional requirements, such as FR_POWER_UPS, FR_REPUTATION_POINTS, FR_EARN_MONEY and FR_SPEND_MONEY. We added to the table with requirements like these in order to cover all the parts of the customer's brief, including the new additions added for the second half of the project.

Once we had changed and added to the table, we added to the explanation report made by the other team in order to explain the new formatting. The other team's report was largely unchanged in its idea, but was rewritten to include full sentences as well as proper paragraph formatting. Additionally, it was modified to include new paragraphs to do with explaining the re-formatted requirements table, as well as talking about our second client interview. These changes were done in order to make the requirements deliverable appear more professional and to explain our contributions to the previous team's project.

Architecture

Original: Arch1.pdf

The first step when taking over the Architecture deliverable was to carefully examine the diagrams and report provided and consider if we should make any changes to the presentation of the architecture. Interim architecture diagrams can be found under the 'Architecture' section of the website, and these are referenced in bold (for example, **[ARCH1]**) to help the reader find these easily on the website and the deliverable.

The Assessment 1 Deliverable included several structural diagrams. The first of these was the 'Final Product UML Diagram'. This was a class diagram designed to display every class in the project. Due to the volume of classes and abstraction used, the diagram was very dense and difficult to read.

There were then several 'sub-diagrams' provided to give the reader an in-depth view of each package. These were much more readable and listed the attributes and methods in each class. However, there were some packages that did not accurately represent the final architecture of the system. This was likely due to them not being updated at the end of the project.

We elected to redraw the 'Final Product UML Diagram' using PlantUML. **[ARCH1]** Since the original diagram was not readable, this was done by studying the code. The result was a simple and comprehensive overview of the system architecture. We then also improved the sub-UML for the package 'game', which was not fully updated. **[ARCH2]**

The Assessment 1 Deliverable contained only one behavioural diagram, the 'Product Use Case UML'. We thought that including a use case would be very useful to capture the requirements of the system and how different elements interact with one another in a comprehensive way. To make the diagram even more useful, we added boxes categorising it into several sections: 'Game_World', 'Menu', 'Pause_Menu', 'Hand', 'Collisions', 'Direction of Travel'. Some of these were divided into further subsections. This allows a reader to better understand which element of the game they are inspecting and the relationship between different elements. **[ARCH3]**

We then wanted to provide more detailed information about the controls in the game. This was done by providing 'zoomed-in' versions of the original use case. **[ARCH3a-d]**
We also wanted to include a detailed sequence diagram to demonstrate the recipes in the game. This was key as the game revolves around recipe making, so having a clear plan as to how the recipe system worked before implementing any further changes was very useful. This allowed us to extend the architecture in a way that made sense with the original code, and avoid implementing unnecessary methods or attributes.

Once we had improved the initial architecture diagrams describing Assessment 1, we were ready to begin designing changes to the architecture itself. The first step was to refactor the code in such a way that it allowed the highest test coverage possible. Initially, a large portion of the game relied heavily on the GameScreen class. The issue with this was that it required an instance of the game to be booted to test many of the methods. To improve coverage and

make testing as easy as possible, we started by refactoring the code to separate as many classes as possible from the GameScreen.

We removed the use of BodyHelper and implemented collision detection exclusively using interaction rectangles, which allowed tests to be written for collision. Much of the customer management code was implemented within GameScreen itself. To combat this, we redesigned the customer code to also extend GameEntity as part of the ‘cooks’ package, which we renamed to ‘players’. CustomerController was also moved into the ‘players’ package, and many of the customer management methods were moved to work exclusively in CustomerController. GameScreen was used only to add and render customers into the game. Finally, there was a requirement to implement an assembly station and a serving station separately. The original code did not follow this, as the components of a recipe were all added onto the serving station and either accepted or rejected by the customer. Working closely with the new architecture design for customers, a new station was created called AssemblyStation, which also extended from Station. The logic that implemented assembly in the original ServingStation class was moved over to the new class, creating two distinct stations. **[ARCH5]**

Satisfied that the design was easily testable and extendable, we began to plan how the architecture might be extended to fit the Assessment 2 requirements.

The new recipes for jacket potatoes and pizzas were easily implemented, with the aid of the improved classes and the behavioural recipe diagrams we had designed previously. (FR_ASSEMBLE).

Two new packages and classes were created to cover the new reputation point (FR_REPUTATION_POINTS) and gold (FR_EARN_MONEY) requirements. Package ‘reputation’ containing class ‘RepPoints’ was designed to allow the player to gain and lose reputation points according to the chef’s progress with serving customers. Package ‘gold’ containing two classes, ‘gold’ and ‘shopItem’ allowed the player to spend their earnings and buy new chefs and stations (FR_SPEND_MONEY), and shop for power-ups.

We chose to implement most of the powerups by creating new classes extending from Station, as Station already had many of the attributes and methods needed to implement powerups. For example, the class SpeedPowerUp extending station made use of the interaction methods implemented in Station. This method was overridden so that an interaction with the station led to the chef’s speed increasing. The teacup powerup and change request powerup, however, were implemented using a new class PowerUpPantry extending from Pantry itself. This was because the methods and attributes in Pantry were more suited to the Pantry class than the Station (for example, picking up items). **[ARCH6]**

We were able to successfully implement most of the new requirements using these new architecture designs as our guide. However, while coding there will always be new challenges that come up and some architectural decisions are made during the implementation process. We made sure to add all new classes that were created into our UML diagram as we went, and finished the project by going through each class manually and making sure it was up to date. This ensures that any future developers who may take over our code will have a comprehensive, updated view of the system and can easily plan

how they might extend the project. The final UML diagram, alongside the in-depth method & attribute class diagrams and the behavioural diagrams we have created are included in the final architecture deliverable.

Method Selection and Planning

Original: Plan1.pdf

The method selection and planning report we inherited was largely unchanged from the previous team. The original content of the report was kept, but was rephrased in order to make the concepts more legible.

In addition to this, we added parts to the report about our deliberation techniques used when deciding on which team to pick for assessment two. Since this was a major factor in our project, we believed it was necessary to include justifications about our decision to move on with “UnderCooked” as a project.

After rephrasing their report, we updated it to include all of the methods we had used that the previous team had not. These included things like the use of Trello, the sailboat technique and Git. These additions were made to keep the report as up to date as possible, since the methods we utilised were different to the team before us.

Finally, the report was missing information on the tools used to aid in the creation of architecture diagrams. We added a section in the report discussing our chosen tool, plantUML, and why this was more suitable for our project than the alternatives.

Risk assessment and mitigation

Original: Risk1.pdf

The risk register has been updated to include risks associated with our team's experience during development. As well as this we have included further product related risks revolving around the user/customer's experience. Similar or duplicated risks (For example R4 and R5 of the original report), have been condensed into a singular risk or removed. After adding these risks, I have then tracked all our new risks, as well as all the prior risks. All prior risks had a secondary owner from our team assigned to the risk. In addition, any irrelevant or incorrect mitigations have been removed.

Website

After taking over their project, we quickly found that the source code for the website was very inaccessible and difficult to build off. Additionally, personal preference toward a website with multiple distinct pages drove us to rewrite the website in our own image. The website we took over had source code that had been formatted in an extremely unfriendly way. Entire CSS files had been condensed to one line of thousands of characters long and some html and JS files had very clear signs of being generated from a third-party website builder.

Bibliography

[1] ISO/IEC/IEEE 29148-2018 International Standard - Systems and software engineering – Life cycle processes – Requirements Engineering, IEEE Standards Association, 2018.