1.0 Introduction

The team looked through the code of the project that was inherited in detail to understand what had already been implemented, and what needed to be done to complete the game. As this was done, a list of tasks was created that had to be completed using github's issue feature. Throughout development issues were solved, and new ones were created, but over time the amount of issues steadily decreased. During team meetings issues were assigned to individuals, who implemented them in feature branches in the github repository. Some issues were focused on implementing new features, whilst others looked at enhancing existing game mechanics and bug fixes. The list of issues was used to help keep a log of the changes and additions to the game.

1.1 Game Management and Logic

The GameManager class has undergone a lot of changes from original code provided to us. One of the first changes made was deciding that there should be only one human player per game. This was because in our opinion a strategy style game with more than one player looking at the same screen would remove any benefits of adopting a strategy as the other player would be able to see what they were doing at all times. Therefore, Requirement 2.1.iv has been changed to be so that the game requires one human player and one or more AI players.

It became apparent that GameManager was not developed with foresight for the rest of the requirements as there was no consideration for timed phases or the use of AI players. The code had been developed with a degree of separation from the Unity game engine which made systematically calling certain parts of the code, such as to keep track of market resource prices, nearly impossible. In order to overcome this, the 'PlayerAct()' method was replaced with an 'Update()' method which gets called every update cycle from a passthrough with the Unity game engine. This method now contains all the game logic for going through the various game states, and is implemented in a sort of state machine like model where one state will trigger another and so on. To make this a little bit more manageable, additional game states were introduced to those defined in Bug Free's requirement 2.2 to allow for more back end logic to take place. This logic was taken from the team's Assessment 2 implementation and behaves in exactly the same way, with noticeable changes being removing all the networking code and the college selection phase. The method 'OnPlayerCompletedPhase()' was added as a way of advancing the state of the game when the player has finished one of their individual phases, and is called from the HumanPlayer class when the player clicks on the end phase button, or through the AIPlayer when it completes all its logic and decisions for that phase.

1.2 Events

The event system that was inherited from the previous developers was very basic, not fully complete and did not contain any actual events. Instead it was a shell that could be expanded upon to take the events system in our own direction. The team made the decision to keep the existing grouping of events, regular and crazy, with the chance of crazy events occurring much lower than the chance of regular events.

It was determined that each event that could occur would have its own unique class, which is where the logic associated with that particular event would be written. These event classes all inherited from an abstract event type (Event). This is so they are forced to implement a method that will get called when the event starts, in addition to allowing us to make lists of all the events under a common type. An instance of all the events are created and tracked in the RandomEventFactory class. It must be noted because of this setup there is now no use for the RandomEventStore class, and so it was removed. This architecture was chosen as it was deemed unnecessary to split up the event selection logic, and the collection of events themselves.

As well as the event classes there are also game objects for each event, used to display UI to the player notifying them that the event has occurred (as stated by requirement 2.5.c). These game objects are all hidden by default. There is a common event script that is attached to each of these game objects, which is used to display them for a number of seconds before hiding them again. When an event does occur, the event specific class makes a reference to its corresponding game object, and using its script calls a method to show and then hide the UI for that event.

At the end of the game round (in the recycle phase of the game manager) a call is made to the RandomEventFactory instance to pick an event to start. This differs from the requirement the previous development team set, which stated that the events should occur in the production phase (requirement 2.5.b). This change was made as the style of events in the game are more suited to the end of the game round, and the team believed that in general the gameplay and experience was improved when the events occurred at the end of the round.

When the RandomEventFactory is called there are two possible cases. One case is that no event will occur, but if the other case holds and an event should occur, there is another set of probabilities which determine whether the event will be regular or crazy. An event is chosen at random from the determined category, and the object associated with that event is referenced to start the event.

It must be noted that the events implemented do not adhere to requirement 2.5.a, which stated that the game must contain events that should change the map in some distinct way. The team chose not to do this as the events that were designed did not necessarily affect the map, and were more orientated towards affecting the players and their resources.

1.3 Abstract Player

The previous development team had a concept of a hierarchy in which player classes were a part of. What was called Player has now been renamed to AbstractPlayer. Most of the functionality has remained the same, with some slight changes and additions. One addition is the abstract method 'StartPhase(Data.GameState state, int turnCount)', which forces inheriting classes to implement a body for this method. The team did this so that each type of player can be notified via this method when the game changes state, and can then apply the appropriate logic with the current state in mind.

The method that acquires a tile now checks whether the player has enough money to purchase that tile, in line with requirement 2.2.1.ii. If the player does have enough money, they will be charged for acquiring that tile and the money will be given to the market. This was done to ensure the player, both human and AI, couldn't acquire a tile without paying for it. Other changes include not being able to acquire the same roboticon twice and also checks to make sure that when a player upgrades a roboticon, they must be the owner of that roboticon. The calculate score formula has changed, which is explained in more detail in part 1.9 of this document.

1.4 AI Player

Similar to the situation faced with the development of the event system, the AI received from the previous developers was incomplete. For this reason, whilst the architectural relations remain mostly unchanged the internal structure of the AI has been greatly altered. These changes have accommodated for the refactored GameManager through ensuring that the AI was built from the ground up to deal with its new execution process. The only concept that relates the current implementation of the AI and that of our predecessors, is the use of a state machine when conducting actions. This is an essential concept relevant throughout the project and therefore it is only natural that the AI would function in the same manner.

Due to the complexity and time requirements of developing an AI which accommodates a concept of proficiency (or rather difficulty levels), it was deemed necessary by the team to abandon this approach of creating rewarding replay value. Instead through implementing a singular AI with actions being made based on 'chance' (implemented through the use of randomly generated numbers), it is our hope to achieve the same measure of replayability. An example of this can been seen in the AIs interaction with the market and whether it would like to purchase resources. Whilst the decision is greatly influenced by probability (which is reliant only on price history), the AI makes a 'gamble' on whether the price will increase or decrease.

Simple probability ensures that the AI is not too efficient in the hopes of producing realistic (meaning more like a human player) and fair gameplay (not being able to predict the market changes with complete accuracy). By developing a AI that makes mistakes that a player can capitalize on the team hope to increase the enjoyability of the game, having a fallible AI will likely lead the player to making more risks in the hope of making the most of any mistake - creating more exciting gameplay.

Whilst the AI functions based on predictions and ‘chance’ it still plays in a logical manner and provides a challenging opponent. This was accomplished through scoring tiles which allows for the AI to easily compare and contrast different choices and choose the direction with low cost and high reward. To ensure that it is possible for the AI to lose (as this may not be guaranteed by 'chance') and function in as consistent manner as possible, these weighted decisions are used along side logic that helps limit the efficiency of the AI without compromising gameplay. Such an example is ensuring that the AI is unable to bulk sell its resources at times it predicts the market price has reached a maxima, as within a dynamic market this could lead to frustrating results for the player (frequently crashing prices and an unstable economic model).

The decisions on trading (i.e. making/cancelling a trade or purchasing a trade) proved difficult to implement in a way that mimicked a human player. As such the logic of trading is based solely on the current and average prices of the resources, which while predictable is functional.

1.5 Market Demand and Supply Pricing

A supply and demand market was required as stated in requirement 2.4.a and was not implemented by the previous team. The Market class was modified to track the total supply to the market which is defined by the total amount of resources mined by both players. It also calculates buy and sell prices of resources by using the incoming supply of resources to the player from tiles, roboticons and roboticon upgrades. The market does not take into account the amount of resources it owns, as it proved difficult to create a model to take this into account, therefore requirement 2.4.b is not completely implemented. As the buying prices are derived from the selling prices, one can ensure that it charges more for it’s resources than it will buy them from the players for (requirement 2.4.c). When the team took over the code much of the market purchasing logic was within the Player (now AbstractPlayer) and HumanGui classes, it has since been refactored to all be within the Market class, as this logic is independent of any player and should be the same for all entities interacting with the market. The GameManager was also updated to call the methods outlined above at the recycle phase as this ensured that the market prices would not fluctuate within a phase - giving each player a fair opportunity to trade, which proved to be important when playing against an AI is it completes its moves far quicker than the human can.

1.6 Player Trading

Trading between players is a feature that was in the requirements that had not been implemented by the previous development team. They made shells of classes that could be used, but in practice the team found it easier to write the system from scratch. This logic is utilised by human players through the market UI, and AI players when they deem it necessary to create, cancel or accept a trade. Once a trade has been created it will stay in the market until either the creator cancels it, or another player accepts it. Once another player accepts a trade, they give the appropriate resources or money to the trade creator, and vice versa, and after this the trade is removed from the market.

All trades require three main pieces of information; a resource type, and amount of that resource and an amount of money. The trade creator specified the amount of the specified resource they are willing to sell, and for what price they wish to sell it for.

1.7 Gambling

Gambling is yet another feature that was implement from scratch due to it not being an immediate requirement for the previous development group. The Casino class is used to manage gambling and allow players to risk an amount of their money to potentially double it. Players can engage in gambling during the auction phase, where they specify an amount of their money they want to risk. The casino is initialised a maximum win percentage, which is directly proportional to the average win rate for the players.

Once a player decides to gamble a random number is generated, and if it is within a certain threshold (determined by the maximum win percentage) the player gets the stake they risked back plus two times that amount in profit. Otherwise, the player will lose the money that they risked. The team added the maximum win percentage as a way of controlling how fair the casino is. The casino satisfies requirement 2.4.d.

1.8 User Interface

The user interface has been implemented in a way that did not make sense given the project is being written using Unity3D. It appeared that the original team developed the UI in such a way that you could take it away from the back end and replace it with another system without having any errors, however given Unity is an all inclusive game engine and there was no requirement to replace the front end, the architecture choice didn't make sense. The original system had a joiner class HumanGui that connected the back end classes and the front end Canvas script, although this is redundant and inefficient it worked for the most part so the decision was taken to leave this as it was and implement any new UI elements in a different way.

New classes have been added to handle new UI features; gambling, player trades/auction, random events and the win condition. These UI elements have scripts attached to them which are used to update elements such as text objects based on some action in the code, for example the CasinoScript class has methods which update the UI images to display the number that the player rolled on during their gamble, and the ScoreBoardScript class has a method to display the winning player's name and score on the popup box.

1.9 Winning Condition

No win condition had been implemented by the previous developers to give the winner of the game. To ensure the winner was calculated at the end of each game a method was created that returned the winner once there are no available tiles left to obtain. The winner is determined by the player with the highest score (as stated in requirement 2.3.c) and hence a method was needed and created to calculate the score based on the land and resources owned by each player (requirement 2.3.b). The score is calculated in the AbstractPlayer class from the player's remaining resources on their tiles, their remaining roboticons and the amount of remaining resources. Hence the player with the highest score when the game ends is pronounced the winner; displayed in a pop-up window in the UI.

1.10 Bug Fixes and Minor Features

The code which the team has inherited from the previous developers did not work as intended in multiple instances. The majority of issues came to our attention after playing the game, however some of them only emerged after an intensive testing. No major features were implemented initially by the team, rather, bugs were fixed and minor features were added to make the gameplay experience more intuitive and enjoyable.

Text which displays the market money was added to allow the player to make informed decisions (requirement 2.6.a). A bug which caused the money spent on tiles and roboticons disappear was also fixed in order to keep our market system closed. Without that fix, both market and the player quickly run out of money to spend. A full list of issues (both open and closed) can be found at https://github.com/Tim020/SUPER-SEPR/issues.

1.11 Non-Implemented Features

During development the team had to make a few decisions which resulted in some of the original requirements being dropped. One of these is changes was to no longer make it a requirement to be able to load and save the game. Making this change actually did not affect the existing code base in any way, as saving and loading would result in additional methods being written, and some existing ones in the GameHandler class would have to be modified. It was unnecessary to implement this requirement purely because our opinion was that the length of the game didn't warrant saving features, and therefore did not require loading either. Another feature not implemented was to render a roboticon sprite on a tile if a roboticon was installed there. The main reason for this was simply because there was no time to include and test this enhancement, and leaving it out would not make a huge impact on the gameplay.