Implementation Report

References to requirements are given in **[ ]** and can found here
Architecture upon receival of the game
Architecture on submitting the game
Architecture plan from Fractal

Food Implementation

Prior to assessment 3, food had already been implemented into the following classes; AnimationAddResources, RoboticonType, LandPlot, Market, NotCommonResourceException and PlotManager. Since then, the food resource has been newly implemented into six classes.

The following changes were made in order to fully implement food as an in-game resource and meet several of the requirements:

- **GameScreenActors** - Added food label to on-screen resource list and plot output details **[1.1.2]**.
- **ResourceMarketActors** - Added buy and sell food buttons in market, setting them to only trigger if the player has the correct money/resources to buy/sell it **[8]**.
- **RoboticonMarketActors** - Added food roboticon texture and added food to roboticon customisation combo box **[6]**.
- **ResourceType** - Added food as a working and valid ResourceType **[7.1.3]**.
- **Player** - Added initialisation of food resource for players with an initial value of 0 **[8.2.1]**, added associated accessor methods for food and added food to generateResources method **[7.2.2]**.
- **TileConverter** - Added food to getRoboticonTile accessor method with an appropriate resourceIndex **[6.1.4]**.

Given that player resources were already implemented into the architecture, the addition of food was already supported by the architecture and therefore did not require any significant changes.

Endgame Implementation

The requirements specify that after all plots have been allocated, each player's score is calculated based on their wealth. The player with the highest score is then declared the winner **[10]**.

Given that a way of implementing this feature wasn't provided within the architecture, we had to extend the architecture ourselves in order to meet this requirement. We decided that both players scores, as well as the winner, were to be presented on a separate screen after the game ends. This required the creation of two new classes, the GameScreenActors class that created the actors that were to present this information, as well as the GameScreen class which presented the actors.

Methods also had to be written that calculated a player's score and find the winner. This required a method to be written in the player class calculateScore, which calculated a score based on the amount of resources a player had **[10.1.1]**. An addition a method in RoboticonQuest was added to find the winner based on their scores due to this class holding the player objects for both players. There also needed to be a method to check that the game had ended within this class, this was done by creating a method that checked whether any of the land plots contained within the PlotManager were empty. This is due to land plots being created upon being acquired, so being null would assume they haven't been acquired.

Overlays

The game that we inherited gave the player access to the market for both resources and roboticons via a different screen. While this would be functionally competent enough, we felt it would be more aesthetically pleasing to just have a simple popup screen within the main game window. This would mean that all of the game's actions would take place within one window rather than opening up different screens for each phase.

In order to implement this we created a class called Overlay. This extended the stage class by placing a rectangle and table within the stage. The information to be displayed and any necessary buttons could then be placed within the stage with the rectangle as the background.

Note that we created the Overlay class during assessment 2 and simply decided to re-use it for assessment 3: this is because the class operates entirely independently of any game engines and simply leverages LibGDX to draw an interface directly to the screen.

Effects

In accordance with the scenario brief, random effects were to be introduced into the game**[11]**. Like the implementation of the end game functionality, the architecture had to be modified as there wasn't any framework for adding random effects to the game. This meant the creation of two entities, PlayerEffect and PlotEffect. Player effects will alter resources within the player's inventory whereas plot effects will provide modifiers for resource yields of plots. These will be implemented using the previously mentioned overlay interfaces, giving a description of the effect to the player once it has been triggered. Their actual effects are triggered with the use of a runnable.

One method for the initializing all of the effects would be to write them in the RoboticonQuest class(the game engine) which is where most entities are created, however this would have made the class far too large. Therefore we created two utilities, PlayerEffectSource and PlotEffectSource where the effects would be written and then implemented. These utilities simply extend an array containing the respective effect objects.

Team Fractal's planned approach to implementing this feature within the architecture was to use an 'event' class. We instead used the classes as described above due to the ease of implementation and the abstract nature of the event class that was presented within the previous architecture.

Artificial Intelligence

Upon receiving the project from Fractal their architecture stated that both Human and AI player classes should inherit from the Player class. However contrary to this they had implemented it such that the Player class was the Human player. So instead of refactoring their solution which worked. AIPlayer inherited from Player as intended and their proposed Human class was discarded. The function "takeTurn(int phase)" was added to Player. This function was left blank in the Player class as it is designed to be polymorphically inherited into AIPlayer. This allows the game to request action every turn, which won't affect a human player's gameplay but will call the AI to action.

Users can choose to play against the AI player or a human player upon starting the game **[4.1.1]**. The AI player takes the first turn, this can allow the player to see how the game can be played. Turns are taken as normal between AI and human player **[4.1.3]**. The AI player completes its turn almost immediately **[4.2.1]**. All requirements associated with the AI player have thus been met.

Gambling Minigame

The player can access the bar screen via the market screen “Pub” where they can gamble with their money through a minigame**[9]**. Team Fractal’s architecture described the implementation of the minigame through a Minigame class. Due to the trivial nature of the dice minigame that we wrote for the dice minigame that was implemented, we felt there was no need for a separate Minigame class. Instead, we have implemented the gambling via the gamble() method within the ResourceMarketScreen. This is because the bar where the gambling takes place is accessed through the market as described in the brief and the requirements**[9]**.

The gamble() method allows the input of a value (that is the money that player will be gambling with) by the keyboard. Numbers are randomly generated between 1 and 6, that corresponds to 6 possible dice values. Depending on player’s and AI player’s dice values “YOU WON”, “YOU LOST” or “YOU DREW” messages are displayed that indicates win, lose or draw condition. Depending on gambling outcome player’s money, gambleMoneyWonCounter (indicates amount of money won in current gambling session), gambleMoneyLossCounter (indicates amount of money lost in current gambling session) and gamleWinLossCounter (indicates number of wins and loses) are updated.

Market Supply/Demand Mechanics

The market prices were currently static so it meant that we had to impose supply/demand mechanics to change the prices **[8.1.2]**. This meant the creation of an algorithm within market class that imposed this feature. The sell price is simply calculated from the amount of resources within the market, the buy price takes the sell price and then applies a modifier to it.

New Market GUI

The ResourceMarketScreen and the RoboticonMarketScreen are now presented within an overlay rather than a new screen. As a result they now extend the Overlay class instead of the screen class. This links to requirement **[1]** as it helps provide a better GUI by making it easier to read.

Random Resource Values for Tiles

The resource yield values for each land plot were set at static values that purely depended on the type of land plot that they were, e.g a cityTile or a forestTile. To add further variety to the strengths and weaknesses that each land plot had we added a random modifier to each land plot**[1.2.1]**. This was implemented via a simple method within the PlotManager class.

TTFonts

This class was created in order to support the implementation of more than one font into the game due to there only being one font currently supported, which was implemented via a skin. The TTFont class simply takes a .TTF font and converts it to bitmap format, the format that libGDX needs to display the font correctly. All fonts are stored within the Fonts class. This utility simply creates and stores each TTFont object as an attribute, rather than storing each individual font within the game engine.

Like the Overlay class mentioned before, the TTFont class was created during assessment 2 and ported over to *Fractal's* game for use in assessment 3: this was possible because, again, it operates independently and does not rely on any game engines in order to function.

This implementation doesn't change or remove any of the aspects of the previous architecture, but instead adds to it. Like the implementation of the overlays, the purpose of having different fonts is purely for aesthetic purposes. Therefore it can be linked to requirement **[1]** as it helps provide a clearer GUI by uses a less monotonous font style.

Features for Assessment 3 Not Fully Implemented: Auctions

Team Fractal's architecture specifies an auction class that implements a bidding system. However this isn't consistent with their requirements as requirement **[8.1.3]** was removed. It is also not possible to implement this feature due to the hot-seat style of the game as two players can't simultaneously bid for a resource if only one is playing at any instance. Therefore we removed the class from our architecture as the need for it was no longer there. All buying/selling is implemented through the market class.