During Assessment 3 there were a few large features that needed to be implemented, these were:

- An AI Player (Requirement 4.1.1)
- An option to chose the number of players (AI or human) (Requirement 4.1.1)
- The minigame (Requirement 9.1.1)
- Trading with other Players (Requirement 8.1)
- Random events (11.1.1)

To implement the AI Player we extended the normal player class, this is useful as it allows us to store all Players in one Array in the Game Engine, this makes things like choosing who's turn it is much easier as you merely have to iterate through an Array. Inheriting the constructor from the Player is also useful as it needs to store the same information i.e. the resources and money the player has. On top of that the AI has been given methods to chose what to do in a given round. Unlike a human player the AI does not need to interact with the GUI so it can just directly call methods in the market and the engine to trade and claim plots.

Choosing a number of Players required a number of changes to the way the Engine stored and treated players. Instead of a fixed set of Players being generated and Colleges being assigned directly to them, a fixed list of all colleges is generated, then an list of players is generated, the size of which is dependant on the amount of players selected. The players are then assigned to a college in the college list. The list of players stores bot AI and regular players for the convenience of moving between them in the game. A new screen was implemented to allow the user to easily select the number of players. The user is taken to the screen before the game begins so they cannot accidentally start a game without selecting the number of players. However, this screen needed access to the engine to tell it how many players were in the game and originally the engine was not initialised until the GameScreen was created. The GameEngine and the GameScreen were also both dependant upon each other to initialise. Because of this the initialisation of the engine and the GameScreen had to be moved to inside the player selection screen and a new method to assign an engine to the GameScreen was created.

When implementing the minigame we decided to keep it simplistic, this was so we could spend more time implementing the core elements of the game. We elected to make simple card flipping game where you spend money and then choose one of three face down cards. When a card is flipped your reward (or loss) is revealed. The game is easy to play and will not distract the player from the overall game. If there is not enough money to pay for the game's fee then the player cannot enter the mini game.

For trading with other Players our original Architecture had an auction class which would take offers and get prices from various players, choosing the best price. However, in a turn-based game and a game that might not have many players we decided that this would not work very well. Instead a Players can directly send offers of resources for a set price directly to a player of their choice. This is much better in a turn based game as it only takes one turn for them to either accept or decline the offer. To implement this system a new Trade class was implemented, the class contains the amount of resources on offer and the price they are being sold for. The Trade can also reference who the trade is to and who it is from.

Trades also contain an execute method, which upon confirmation from the receiver will move the resources and money around. Trades are created in the market and then stored in an Array in the Engine. Since they are in the Engine it is very easy to check if the current player has any pending trades at the beginning of a new phase, as you just need to iterate through the array and check if any of the trades are aimed at the current player. When the trade is either accepted or declined it is removed from the Array in the engine so it does not appear again.

To create the random events which occur every round in the game, we decided to create an abstract class called RandomEvent. We did this because there are many methods which are required to be implemented across random events, and methods which behave in exactly the same way across all random events and can therefore be shared via inheritance. By doing this, it gave us an easy way to create new kinds of random events, without having to reconfigure lots of code across many files in the project. In the sub-classes Earthquake and Malfunction, the back-end effects are described, along with the messages which are passed to the designated random event overlay in the GUI. To prevent the same type of random events from occurring too frequently, a cooldown system was implemented which meant that after an event occurred, a new event of the same type could not occur for x turns. The random events are stored in an ArrayList in the GameEngine class.

There were also other smaller additions implemented:

Resources needed to affect the rate of production (Requirement 7.1.4)
When a tile produces resources it consumes food and energy, if the player does not have enough food or energy resources are not produced

The Market needs ore to generate Resources (Requirement 6.1.2)
A method has been implemented so that the market will test the amount of ore it has and generate new roboticons based on that.

Changes

Class	Change	Side effects	Justification
Engine	Store players in a list which is initialised with a size selected by the user.	Next player class had to be refactored to iterate through a list that does not have a set size.	Allowed us to vary the amount of players in the game and chose to have an adjustable number of AI players.
Engine	Store colleges in a list in the engine class.	-	Assigning players to a college is much easier if they are stored in a list as you only need an index.
Engine	Refactored	-	Made the code easier to read,

	nextPhase() method into a switch statement and moved parts of it into their own methods.		maintain and update.
Engine	Changed the nextPlayer() method to iterate through the player list and deal with AiPlayers.	More complex code.	The game can now deal with more than 2 Players and also allows the player to play against an AI player. (Requirement 4.1.1).
Drawer	Added the ability to draw a roboticon on a tile.	-	Gives the player feedback after installing a roboticon.
GameScreen	No longer initialise engine in GameScreen, but assign it after it is made.	-	The engine is needed in the PlayerSelectScreen before the GameScreen is initialised.
Market	Put the auction in the market space and added buttons to move between them.	-	Allows us to keep all trading related elements in one place on screen. Whilst separating player and market trading by preventing them from appearing at the same time to avoid confusion.
AbstructAnimation, IAnimation, IAnimationFinish	Code from previous assessment, add animation interface for game.	-	Allowed us to continue use the familiar animation code.
AnimationTileFlash, AnimationPlayerWin	Add new animations.	-	A few animations for the game, to make it more [...]

Player	Show player number method.	-	The old method use the player number as ID and the index for reference, which increases the unnecessary memory usage. Now the player ID is zero based and the new player number method return the value of ID + 1.
MiniGameScreen	Added the mini game screen and integrated to the game engine	-	This is for player to gamble, it cost 20 per game. It may get +100 money or +1 roboticon or get nothing. The price will transfer to the player immediately. If player do not have enough money then they cannot enter the game.
AiPlayer	Added AiPlayer extends current Player class.		Its process phase method is called automatically on phase end, to make human player play with ai player.

As the Game passed all requirement testing, we are confident that all features required for assessment 3 have been fully implemented.