To implement the ability to have up to four players we first had to change the way in which a new game was initialized. Initially, the new game function took a boolean parameter to determine whether or not you would be playing against a human opponent or an artificial opponent. Since we needed to allow up to four players (Requirement 4.1.4) the function was changed to take two integers as parameters, one for the number of human players and one for the number of AI players.

Within the new game function, we had to change the way the list of players was created. Before the change, it only had an "if statement" testing the boolean parameter and filling the list according to its value, we modified it add the number of players and AI players with for loops to the player list. Doing so allows us to give the player a choice in how many human and artificial intelligence players they would like to play against.

In the main game loop, the next player function was made to iterate through the list of players and give the appropriate player control of the game. However, the old function only worked for a list of size 3, with two players and index 0 empty. Now it works with any size list and just increases the index of the current player until it reaches the end of the list and then goes back to zero.

To allow the player to utilise the new functionality, we implemented a new screen for the player to be allowed to chose how many players to have in the game. The new screen is made up of 2 numbers with buttons to increment and decrement them next to them. The two numbers represent the number of AI and human players. The player can use the buttons to modify the value of the numbers. However, they cannot choose a negative number of players, and they cannot have more than four players total as the increment/decrement buttons disable if they would make the number invalid. Finally, there is a confirmation button to lock in the number of players and start the game. This button is disabled until a valid number of players have been selected, and then it starts a new game, allowing us to easily control the number of players, ensuring the game can only start with a valid number. The fact that the buttons give visual feedback when they are enabled/disabled can give the player an idea of what configuration of players they can legitimately play with. The buttons are stored in a separate class, as detailed in the architecture.

The last thing to do when implementing “up to four players” was to take the player to the player selection screen. This function was made more complicated by the fact that in the original code a new game was initialized by the main menu screen. However, the number of players is required to initialize the game, which is not decided until the player selection screen. This meant that the initialization of the new game had to be postponed so that it occurred when the number of players was chosen.

The other new requirement was to implement a capture the chancellor minigame (Requirement section 12). We decided to go for a Pokemon style minigame in which you had to defeat or capture the chancellor in a ball.

A lot of the code for the implementation of the minigame was for the GUI, a number of classes were made to generate the animations involved. These include:

- The TypeText class to create a typing effect for the dialog text to show up on the screen for the minigame.
- The WhiteStrip class which displays strips across the middle of the screen, behind the Chancellor avatar.
- The WildChancellorAppear class which displays and controls the chancellor and how the player interacts with the game.
- The CaptureData class which connects to “capture/data.json”, the configuration file for all the battle related settings.
- The ResourceDelta class which provide a uniform interface for batch manipulate players resources at once.

The WildChancellorAppear class also handles the logic for the minigame. It takes information from the capture data class, as described in the architecture document, to decide what elemental type the chancellor is and the ways in which the player can fight him. It is also in charge of handling all the on-screen effects and the time setting for the minigame.

The minigame appears randomly. During gameplay there is a chance that after buying roboticons the chancellor will appear. There will be 15 seconds for the player to choose whether to fight, capture or run from the chancellor (Requirement 12.2.1). If they pick fight they will be given a choice between elemental attacks and should choose the one that would be most effective against the type of chancellor they face. If they defeat him they’ll get some resources, if not they’ll lose them. If they choose to capture the chancellor they will throw a ball at him and have a chance to capture him and get a large resource bonus for doing so. Finally they can run away and avoid it entirely. Fighting costs 10 money, capturing costs 20 and running away is free. This allows the player to chose to take a small risk and get a small reward. As it is just a minigame we didn’t want it to have too much of an effect on the overall outcome of the game so the values involved are only small.

To ensure the minigame appeared at the right time we edited a state in our loop based state machine. It will now assign a random boolean value and if it is true the minigame will happen. This means the player isn’t faced with it every round and makes the game unpredictable and more fun to play.