Top Right Corner

Implementation report

How our code implements our architecture

[Link to concrete architecture / UML class diagram for our code:
https://sepr-topright.github.io/SEPR/documentation/assessment3/images/class-diagram-2.png
Abstract architecture diagram:
https://sepr-topright.github.io/SEPR/documentation/assessment1/deliverables/Arch1.pdf]

Changes were made from the original abstract architecture. Many being additions to the original abstract architecture, mostly consisting of details relating to the implementation of the program and extra classes. However some changes were made to the original classes from the abstract architecture.

Both the HumanPlayer and AIPlayer class are not present from the current implementation. As this was due to a requirements change where by there was no longer a requirement to have a human player and an AI player in our game. Due to this we no longer required to have a Player class with both HumanPlayer and AIPlayer subclasses and instead could have a single Player class. The Inventory class, PlayerInventory class and the MarketInventory class are not present from the current implementation. Instead the MarketInventory has been combined with the Market from the abstract architecture to create a new Market class with a built in inventory. While the PlayerInventory has been combined with the Player class from the abstract architecture to create a Player class with a built in inventory. As a result of both these changes, the Inventory class is no longer required and is not present in the implementation. Pllease see the last section of this document for more relating to this change. The Plot class from the abstract architecture is now named LandPlot in the current implementation. There was an addition to the implementation called RandomEvents, which was added to satisfy requirement 1.25 which needed random events implemented in the game. If we consider the MarketInventory and PlayerInventory classes to now ba a part of the player and market classes respectively then we can see that all of the associations from the abstract architecture are still present.

How our code implements our requirements

Our requirements have been updated since assessment 2. The requirements, the changes and the reasoning behind those changes can be found here:
https://sepr-topright.github.io/SEPR/documentation/assessment3/Requirementschangesforassessment3.pdf

Requirements 1.1 and 1.2 are satisfied by the code in the nextPhase method in the RoboticonQuest class (which ties together all the other components) as it ensures that all players get to buy plots, buy and place roboticons before the next player is given their turn and that all players get exactly one turn before moving on to the shared market phase. It also ensures that once all plots have been acquired the game does in fact end.

The code in the newGame method in the GameScreen class reads in the map data from a file (which specifies that there should be 84 plots) and then calls the setup method of the PlotManager class that creates an array of LandPlot objects that store all data relating to a plot (owner, specialism etc). The code in the getPlot method of the PlotManager class that returns the LandPlot object associated with a given tile on the screen and is called when such a tile is clicked. While the tileClicked method in the GameScreenActors class decides what to do when a given tile is clicked. Through all of this code **requirement 1.3** is satisfied as the game supports well over 16 plots that the players can interact with during gameplay.

Requirement 1.4 is satisfied by the owner attribute of the LandPlot class and the getOwner and setOwner methods. As well as the code in the clickListener attached to the buyLandPlotBtn TextButton object, that is an attribute of the gameScreenActors class, that checks to see if a plot of land that is clicked is already owned when a player wants to buy it and if it is not sets the owner attribute to that player. The nextPhase method in RoboticonQuest takes the game through a number of phases it enters phase 1 one time for each player in the game per round. When in phase 1 the screen is set to the GameScreen. When in the game is in phase 1 the tileClicked method in the GameScreenActors class that is called whenever a plot/tile is clicked causes a menu on screen which contains the buyLandPlotBtn. Through these mechanisms **requirement 1.11** is implemented. When in phase 3 of the game and a plot that is owned by the player is clicked a menu is opened that contains a drop down menu that lists the number of roboticons with ore, energy and food robitcons (contained with installRoboticonTable, see case 3 in titleClicked). One of these items can be selected and clicked and then a confirm button that is also a part of this menu (installRoboticonBtn) can be clicked. Code is executed when that button is clicked to ensure that selected roboticon type is placed on the plot in question only if the plot does not already have a roboticon on it (see the clickListener that is added to installRoboticonBtn in the bindEvents method), thus **requirement 1.15** is met.

Requirement 1.5 is satisfied by the money, ore, energy and food and roboticonList attributes of the Player class as well as their associated getter and setter methods. Similarly, **requirement 1.7** is satisfied by the food, energy, ore, roboticon attributes and their associated getter and setter methods in the Market class.

Requirement 1.6 is satisfied by the initial value of 100 given to the money attribute in the Player class and **1.8** by the calls to the setFood, setEnergy, setOre and setRoboticon methods in the constructor of the Market class.

Each element in the productionAmounts array in the LandPlot class describes how many units of a given resource a plot is capable of producing in one turn. These values are set via LandPlots constructor which is called by the createLandPlot method in the PlotManager class (see above). The createLandPlot method determines that city tiles/plots are best at producing food, forest tiles/plots energy and water and hill tiles/plots food. Through these mechanisms **requirement 1.9** is satisfied.

Requirement 1.10 is satisfied by the nextPhase method in RoboticonQuest. After every player has finished their turn this method checks to see whether all plots have been acquired and the last player has had their turn (see case 6 in the method). If so, the game goes to the GameOver screen, ending the game.

The nextPhase method in RoboticonQuest uses the variables newPhaseState along with a switch case statement to decide which code to execute for each phase. Each successive call causes newPhaseState to be incremented causing the code for the next phase (the next case in the switch case statement) to be executed. The case for purchasing and customising roboticons (**requirements 1.12 and 1.13**) comes right after the case for buying plots causing players to enter into this phase of the game after they are acquired a plot and clicked the next button (as required). The screen is set to the RoboticonMarketScreen which adds all the widgets stored in RoboticonMarketActors to the screen. RoboticonMarketActors creates a button stored in the variable buyRoboticonsButton that and when clicked code is executed that first checks to see if the player has enough money to buy a roboticon and if they do the purchaseRoboticonsFromMarket method in the Player class is called which increases the quantity of roboticons and reduces the amount of money stored in the player's inventory as well as reduceing the quantity of roboticons in the market's inventory. Thus **requirements 1.12 and 1.13** are satisfied whilst ensuring that **requirements 1.5 and 1.7** are are also satisfied.

Cases 2 and 3 (that correspond to the roboticon purchasing / customising and roboticon placing phases of the game) of the switch case statement in nextPhase create a new animationPhaseTimeout object. This object is added to the screen and displays a timer on the screen, the tick method in animationPhaseTimeout is called once per second to update this timer and when it reaches 0 the callAnimationFinish method is called that calls the nextPhase method taking us to the next phase. Therefore when the player has ran out of time they are taken out of roboticon purchasing / customizing and roboticon placing stages and in this way **requirement 1.14** is met. Case 4 causes a call to the generateResources method in RoboticonQuest to be called that calls the generate resource method of the current player which calls the produceResource method for each plot that the player owns three times (one for each resource type) and adds the resources produced by the plots to the player's inventory, each call causing the plot to generate the correct quantity of that resource depending on the customisation type of the roboticon it has placed on it. Therefore **requirement 1.16** is met.

nextPhase ensures that each player completes the roboticon purchasing / customising and placing stages before moving on to the shared market phase. See case 5 that calls nextPlayer so long as the current player is not the last player which then falls through to case 6 that will then reset the game to phase 1. If the current player is the last player in case 5 only then is it possible to access the market screen (/ the market). Therefore all players must take their turn before the market can be accessed and **requirement 1.17** is met (note that case 5 comes after case 4 that causes resources to be generated). The market screen class adds all the widgets defined in the MarketScreenActors class to the screen. This includes the following widgets: playerToPlayerResourceDropDown, playerToPlayerSellerDropDown, playerToPlayerBuyerDropDown, playerToPlayerPriceAdjustableActor, playerToPlayerQuantityAdjustableActor, playerToPlayerTransactionButton. That players can use to sell resources from one player to another. When playerToPlayerTransactionButton is clicked completePlayerToPlayerTransaction is called which uses all of the data defined in the widgets above and calls the sellResourceToPlayer method of the selling player object. The sellResourceToPlayer method ensures that the buying player has enough money and the selling player enough of the given resource and if so transfers the money from the buying player to the selling player and the resources from the selling player to the buying player. Therefore (the alternative requirement for) **requirement 1.18** is met. Similarly this class contains the widgets: marketResourceDropDown, marketPlayerDropDown, marketQuantityAdjustableActor, marketBuyOrSellDropDown, marketTransactionButton. When marketTransactionButton is clicked completeMarketTransaction is called which uses the data defined in these widgets and calls sellResourceToMarket or buyResourceFromMarket method on the selected player object depending on whether buy or sell was selected using marketBuyOrSellDropDown. buyResourceFromMarket and sellResourceToMarket both transfer resources and money between the player and market and therefore implement **requirements 1.19 and 1.20**. This class also contains widgets that allow the user to select a player to gamble by selecting a bet they want to place and then placing that bet (gamblingPlayerDropDown, gamblingAdjustableActor, gambleButton). When gambleButton is clicked the playerGambled() method in the ResourceMarketActors is called which then calls the playerGamble method from the Market class and passes it the player and bet specified by gamblingPlayerDropDown and gamblingAdjustableActor. playerGamble ensures that the player has enough money to place the specified bet and if so it randomly decides whether they win or lose. If they win the bet is added to the money in their inventory otherwise it is removed. Satisfying **requirement 1.24**.

The nextPhase method ensures that not all plots have been acquired after the shared market phase is over by calling the allOwned method from the PlotManager class (that checks each LandPlot object in the game to see if it has been acquired by some player and if at least one hasn't it returns false otherwise it returns true). If not all plots have been acquired then the game reverts back to phase 1 (i.e. the next player starts their turn), thus **requirement 1.21** is satisfied. If all the plots have been acquired then the screen is set to the GameOverScreen. The GameOver screen adds of all the widgets in the GameOverScreenActors class to the screen. This includes labels that display all of the player's final scores that are calculated as the amount of money the player is in possession of plus the market value of all resources that they own. This calculation is carried out by the getScore method in the player class. As such **requirement 1.22** is met. There is also a label that displays the winner, the getWinnerText method in ScoreComparissons is used to determine what to place in this label. If both players have the same score then text that states that they drew is returned other wise text that states that the player with the highest score won is returned. Therefore **requirement 1.23** is also satisfied.

The RandomEvents class implements all of the logic for the random events. The roboticonIsFaulty method randomly decides whether or not a given roboticon should explode when it is created and is called when the installRoboticonBtn in the gameScreenActors class is clicked if the given roboticon is deemed to be faulty it is removed from the player's inventory. Similarly the tileHasChest method randomly decides if a given tile has a treasure chest on it when it is acquired by a player (when the buyLandPlotBtn in the GameScreenActors class is clicked). If a given tile is deemed to have a chest on it then the amountOfMoneyInTreasureChest method in the RandomEvents class is called that adds a random amount of money to the player's inventory. The geeseAttack method is also called when buyLandPlotBtn is clicked and randomly decides if geese will steal some of a player's resources if so then getResourceStolenByGeese is called to randomly decide which resource is to be stolen along with geeseStealResources which removes half of the quantity of the given resource that the player has in their inventory. Through these mechanisms **requirement 1.25** is implemented

Requirement 1.26 is implemented as the owner attribute of LandPlot is not given an intial value when the LandPlot object is instantiated and is not set until plots are acquired (in the manner described above) and thus plots initially have no owner.

The attemptToProduceRoboticon method in the market class can be used to make the market produce a roboticon in exchange for 4 of its ore (so long as the market actually has at least 4 ore). This method is called when players click the produceRoboticonButton in the RoboticonMarkeActors class that is on screen during the roboticon market phase. Therefore **requirement 1.27** is satisfied.

<u>Report of any significant changes made to the previous software (and new data structures/algorithms)</u>

We implemented a class RandomEvents this contained the code for 3 random events (see the previous section for an explanation of the events). We chose to implement a new class as there was no logical place to put the code in the existing project. This class contained the methods tileHasChest and amountOfMoneyInTreasureChest in order to implement a random event where a player discovers a treasure chest so receives money. Also in the class was the roboticonIsFaulty method which decides whether a roboticon is faulty. Finally geeseAttack, getResourceStolenByGeese and geeseStealResources in order to implement the random event of geese stealing a player's resources. These were implemented in order to satisfy requirement 1.25.

We completely reworked the ResourceMarketScreenActors class to implement a shared market screen which allows selling to and buying from other players as well as to the market simultaneously for both players in order to implement requirement 1.18. Furthermore we added a gambling system where the players can increase their money satisfying requirement 1.24 which also required a method playerGamble in the Market class. Although this class was completely reworked it had little effect on the overall system as all this class does is create widgets to be added to the screen by the ResouceMarketScreen class.

A major change we made was to finish implementing the food resource (it was already partially implemented) as it was required to fully satisfy many requirements. To do this we added a food enum to the RoboticonType class so you could create food roboticons. We then modified the get CustomisedRoboticonAmountList method so that it would also return a string that listed the number of food roboticons the player had (the strings returned by this method are used in a drop down menu to choose what kind of roboticon to place on a plot so this was required so that food roboticons could be placed) . We then added code to RoboticonMarketActors to store the food roboticon texture for the market screen (so that players could see it when trying to purchase a food customisation) in the variable food_texture to be displayed on screen when players were looking at food roboticons. We also changed GameScreenActors adding a variable storing the food roboticon tile textures and code assigning an image to the variable so that when food roboticons are placed they can be seen on screen.

Another major change to the previous software was implementing a game over screen to be displayed when all plots of land had been acquired in order to satisfy requirement 1.10. To implement this we added a method to the plot manager class allOwned which checks to see whether all plots have been acquired and then modified the nextPhase method in the RoboticonQuest class to check if all plots are owned before going on to the acquisition stage and if they are all owned go on to the game over screen instead. For this to work we created the GameOverScreen and GameOverscreenActors classes to be displayed at the end of the game. We created these two classes to match the way that Team Fractal had implemented all the other screens by having a screen class that implemented the actual screen and an actors class that created all of the widgets/actors to be placed on that screen. We also added a method getScore to the player class to calculate the score as stated in requirement 1.22. Finally we created a class ScoreComparisson which contained the getWinnerText method in order to work out which player had the highest score and return a string that stated which player had won (or if all players had drawn) that used the method allPlayersSameScore method to work out if all players had the same score. This text was placed on the game over screen in order to satisfy requirement 1.23. We created a separate class for this purpose as we didn't feel that this code belonged in any of the other classes.

We also added the getCustomisedRoboticonAmountList to the player class and in the GameScreenActors class changed all references of getRoboticonAmountList to getCustomisedRoboticonAmountList in order to stop players from being able to place uncustomized roboticons (to satisfy requirement 1.15). As instead of returning an array of strings describing the quantity of all types of roboticons the player had that was to be used in drop down menu that allowed players to select what roboticons to place on a plot it only described the quantities of customized roboticons that they had.

Finally we had to give the market the ability to produce roboticons in exchange for some of its ore as listed in requirement 1.27 we did this by adding the oreRoboticonConversion rate attribute and attemptToProduceRoboticon method added to Market class. The attemptToProduceRoboticon method checks to see if the market has at least as many ore as specified by oreConversionRate and if so adds 1 roboticon to the markets inventory and removes as many ore as specified by oreConversionRate. We then added code to RoboticonMarketActors to create a TextButton that when clicked called the attemptToProduceRoboticon method from the market class and added this button to the screen in the RoboticonMarketActors class.

Features required for assessment 3 that have not been implemented

All features required for assessment 3 have been implemented (i.e. all of our system requirements have been implemented). Requirement 1.18 was not implemented, it was a risky requirement and instead its alternative requirement was implemented as we felt that a complete auction system would be overly complex.

Additional changes we would have liked to have made and the more minor changes we made

We would have liked to create player and market inventory classes and moved all of the inventory handling code out of the player and market classes and into these classes in order to further modularise and break down the project. We also would have liked to refactor more of the code as some methods are ridiculously long and there are sometimes multiple methods that do the same thing such as the getResource and getFood/Ore/Energy methods in the Player class. We especially would have liked to rewrite the entire RoboticonMarketActors class as it is a complete mess with massively long methods, stupid ways of laying out elements on the screen (many, many calls to add with no parameters simply to create empty table cells) and many inefficiencies such as creating a new label to display the player's stats (and destroying the old one) every time data or the window size is updated. *We feel justified in our decision to not make these changes* as we had a limited amount of time and a summative exam to revise for so we chose to prioritize the implementation of new features that were required to complete the project and the complete overhaul of the automated tests (which were almost nonexistent when we picked up the project).
A list of the more minor changes we made can be found here:
https://sepr-topright.github.io/SEPR/documentation/assessment3/Minorchanges.pdf