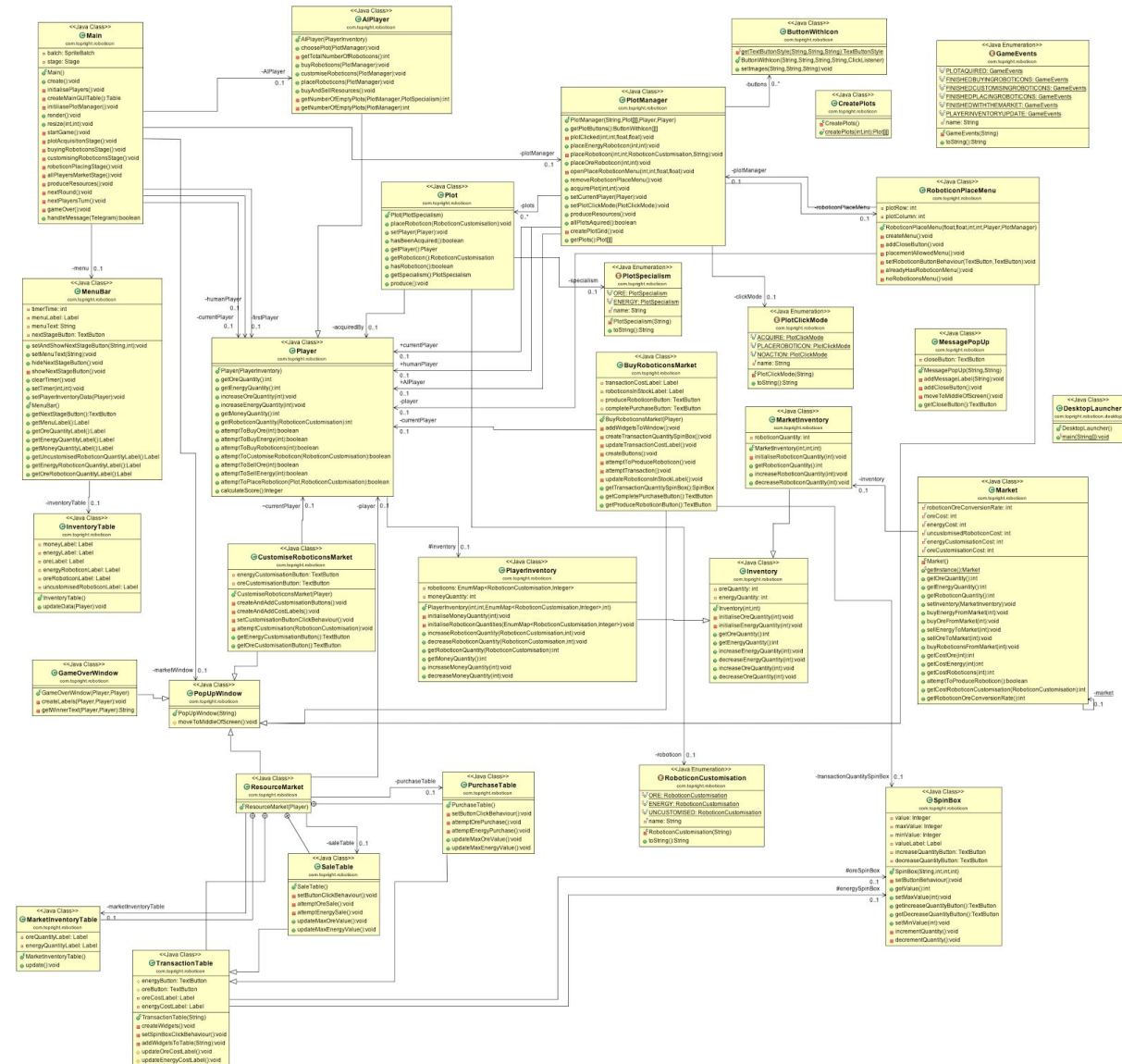


Top Right Corner

Architecture Report

Concrete Architecture

We created a UML 2 class diagram to represent our concrete architecture. This used the UML class diagram notation and was built using the eclipse plugin [ObjectAid](#).



[Link to full size image](#)

How the concrete architecture builds upon the abstract architecture

Several changes were made during the move between abstract and concrete architecture. Many of them were additions to the original abstract architecture, mostly consisting of details relating to the implementation of the program. However some changes were made from the original classes from the abstract architecture. The main change was the removal of the roboticon class from the concrete architecture.

The roboticon class was removed as it was thought to be inefficient to give roboticons its own class and store them separately from the players and the plots. Now in the concrete architecture the roboticons are stored in a variety of ways. In the MarketInventory class there is now a variable for storing the number of uncustomized roboticons, much like PlayerInventory which also stores the number of uncustomized roboticons as well as the number of customised roboticons along with their associated types. While in the Plots class each plot tile now has a variable which stores whether there is a roboticon on it, and if so what type it happens to be. Another change from the original abstract architecture was the addition of costs in the Market class. This gives the market the prices which it should be buying and selling both resources and roboticons at, a feature neglected in the original abstract architecture. The Market was implemented as a singleton class because both players interact with the market and they should only ever interact with the same instance of the Market class as otherwise one player may be able to buy resources that are unavailable to the other (which would be unfair).

Among the additions made to the implementation of the game include classes such as the Main class. The Main class is where the program begins and contains many other implementation related items. Notable examples include the startGame() method where the market, players and their inventories, the map and the GUI are be initialized and setup. Other additions to the architecture include the PopUpWindow which is a class used as a base to build many other graphical windows which include; CustomiseRoboticonWindow, GameOverWindow, RoboticonPlaceMenu, BuyRoboticonWindow and the ResourceMarket. CustomiseRoboticonWindow where the player can customise uncustomized roboticons to collect resources of the selected type. GameOverWindow is used at the end game scenario when there are no empty plots remaining and presents the players scores and displays who the winner of the game is. RoboticonPlaceMenu is used by the player to select which customized roboticon they wish to place on a tile. BuyRoboticonWindow is where players are given the option to buy new uncustomized roboticons. ResourceMarket is a window which allows people to buy and sell to and from the market. The MenuBar and InventoryTable are two other additions and are used for the graphical menu at the top of the game window which are used to relay information to the player. Information such as money and resources available, what phase the game is currently in, and the next turn button.

Use cases

In the abstract architecture we created a number of use cases describing the steps a user might take to perform various tasks and then created UML diagrams (using Lucidchart) to visualize the behavior of the system and the interaction of its components according to these use cases. The use cases implemented many of the system requirements and helped to show the correctness of the model as it supports the necessary communication links between components. In the move to concrete architecture we updated the use cases to reflect the new architecture to ensure that these use cases still implemented the system requirements and still hoped to show the correctness of the model. These use cases can be found here: <https://sepr-topright.github.io/SEPR/documentation/assessment2/UseCases.pdf>

How the concrete architecture satisfies (or fulfills) the initial requirements

The Main class satisfies requirement 1.2 and 1.21 with the method nextPlayersTurn() which ensures each player has a one turn a round, while nextRound() ensures the next rounds start after everyone is finished with the market. Requirement 1.10 is satisfied by Main as the gameOver() method is called once PlotManager finds that there are no available plots remaining. Main also satisfies requirements 1.11, 1.12, 1.13, 1.15 and 1.17 with its methods plotAcquisitionStage(), buyingRoboticonsStage(), customisingRoboticonsStage(), roboticonPlacingStage(), allPlayersMarketStage() being called at the start of their respective turns. Main satisfies requirement 1.16 just after the roboticons placement round where the method produceResources() is called.

Requirement 1.3 and 1.26 is fulfilled by the Main method `initiliasPlotManager()` which setup the map with unacquired plots of size defined within the method of an area larger than 14 tiles.

The `PlotManager` class satisfies requirement 1.10 as if all the plots are acquired then the method `allPlotsAcquired()` returns true, signalling for the game to come to an end. It also fulfill requirements 1.11 with the method `acquirePlot()` which allows the current player to acquire any free plots. `PlotManager` using the method `placeRoboticon()` allows the placement of a roboticon on an empty tile thereby fulfilling requirement 1.15.

The `Plot` class satisfies requirement 1.4 as each instance of `Plot` contains the method `getPlayer()` which returns which player currently owns the plot. Requirement 1.9 is fulfilled using the `getSpecialism()` which returns the resource that tile is most proficient at making when it comes to producing resources for the players.

The `Player` class satisfies requirements 1.13 and 1.15 with the methods `attemptToCustomiseRoboticon()` and `attemptToPlaceRoboticon()` respectively. These allow the player to customise an uncustomized roboticon, given the player has sufficient cash, and to place a customised roboticon given a free plot is available for it to be place on.

The `Player` and `AIPlayer` class both satisfy requirement 1.1 as they both allow for a human player to play against a computer controlled (AI) player.

The `PlayerInventory` class satisfies requirement 1.6 with the method `initialiseMoneyQuantity()` which assigns each player a small amount of money at the start of every game. Both the `PlayerInventory` and `Inventory` classes fulfill requirement 1.5 by `PlayerInventory` containing variables to store the quantity of roboticons along with their type, as well as the amount of money using methods to alter their quantities such as `increaseRoboticonQuantities()` or `decreaseMoneyQuantity()`. While `Inventory` contains variables for storing different types of resources such as ore and energy, along with method for altering their values such as `increaseOreQuantity()` and `decreaseEnergyQuantity()`. The `Inventory` class satisfies requirement 1.8 with its methods for initialising the market with the required number of resources at the start of the game such as `initialiseEnergyQuantity()`.

The `Market` class satisfies requirements 1.19 and 1.20 with its various buy and sell methods which allow players to either buy or sell resources to and from the market place, example methods include `sellEnergyToMarket()` and `buyOreFromMarket()`. The class also fulfills requirement 1.27 with the method `attemptToProduceRoboticon()` which will create a new roboticon if the required resources are available on the market. `Market` also satisfies requirement 1.12 with the method `buyRoboticonsFromMarket()` which allows players to purchase roboticons if they have the required amount of money.

The `MarketInventory` class satisfies requirements 1.7 and 1.8 with its method `initialiseRoboticonQuantity()` which is used at the start of the game to populate the market with the required roboticons and then continuing to keep track of the number of roboticons available using the `roboticonQuantity` variable within the class.

The `MenuBar` class satisfies requirements 1.14 and 1.21 with the method `setTimer()` which initiates a countdown during the rounds which are timed and by having the button `nextStageButton` which allows the players to indicate they are finished with their turn.

The `GameOverWindow` class satisfies requirement 1.23 as it produces a text window and reports the winner using the `createLabels()` and `getWinnerText()` methods.

Please note: `ResourceMaket` contains four inner classes: `TransactionTable`, `marketTable`, `purchaseTable` and `salTable` (`purchaseTable` and `saleTable` are subclasses of `transactionTable`). These classes are inner classes as they are not needed anywhere else in the code but allow us to more clearly organise the code in this class by separating it into distinct logical chunks. The `saleTable` and `purchaseTable` classes share a lot of common attributes which is why they both inherit from a single `transactionTable` class.