SENG201 Project Report S1-2021

Ben Malthus - bma194@ - 22306875,
Kvie Nguyen - kvn17@ - 93014463

Structure of Application and Major Design Choices

RandomEvent Code, as interface

Decided to make the RandomEvent code an interface, instead of having weather, pirates, sailors each being a subclass of RandomEvent because they aren't really the same thing, they are related objects but they support similar functionality so I wanted to add an interface so we can rely on that functionality. Either would have worked.

SubMenu's as a Class for CMD UI

I made two classes to represent often reused functionality in the CMD UI. Both "Option" and "ListOption" are classes that present the user some text (option/s) and then use a scanner to get user input, they have methods to validate the input against a regex, passing program control to a handleOption method once valid input is received. The ListOption version is used when the input requested from the user is choosing which value from the list they want. I made extensive use of these, making a private subclass for each menu in the game, It worked well. (See uml_ui_cmd.png) in submission.

Items, PricedItems, Ship Storage (StorageList)

I spent a huge (too much) effort on working out how to track items that could be purchased / sold / stored etc. I ended up with:

1. Base **Item** class (with Enum for type eg CARGO, WEAPON, UPGRADE etc) to represent objects, including intangible objects like a rescue reward or Wages.
2. Stores (and the Player's transaction list) didnt store the item directly, but stored lists of **PriceItem**, which, as it sounds is a class, to associate an Item with a price (and an island as needed for records). I didn't want one Item class with a price, because a single object can have multiple prices, at different stores, for different purposes eg a forsale price and a forbuy price for the same item
3. Ship storage was dealt with using **StorageList**'s, a ship would have one of more of these objects, for each type of storage they have eg CARGO, WEAPON, UPGRADE etc. An eg Cargo storage list could only accept cargo objects. Each storage list managed checking if items existed, managing space (since items could take up more than 1 unit of space, ie can't just use List.size(), adding and removal of items to that list. This made upgrades easy, eg if a ship was upgradable (it had an StorageList of type Upgrade) and say if an upgrade was a cargo bay extension, I could just add a new storageList to the ship

I was really happy with this structure, although I probably should have encapsulated StorageList further and instead of having Ship have an List of StorageLists, one for each type of storage they have, I could have made the StorageList class encapsulate this further, maybe I'd call it Storage given it represented multiple lists.

Player, World, Ship, Island, Routes

I made a class for each of these and largely split objects into two.

1. The "World" contained all the islands, routes and methods to generate the world (fixed setup), and get physical world layout eg getRoutes(Island). (Island owned the Store object)
2. The Player object, owned the ship, and owned actions with purchases / sales, money etc

The game object owned time and coordinated everything, talking to the UI

Graphical Application (GUI)

This is the first time I have learned to do some architecture for an application using Swing. Swing is quite useful because it automatically generates the code for us, just by dragging and dropping components from the Design tab to the screen. However, I have had some troubles in modifying the code so that it could follow what was implemented in the Command Line Application, I spent a lot of time on getting familiar with all the Swing methods, such as how to display a JList showing the list of routes, ships or items; how to get the user choice after they clicked the radio button.
About the GUI implementation, I made a Screen class, which is the main class to manage all the function calls from other screens. This class contains all main methods to display other screens in the application. Other screens have been created as subclasses of the

Screen class. This class made it easier to move from one screen to another, just by creating a new screen and passing in the instance of the game.
The GUI was tested just by running the application and trying to find bugs in it. This part was quite frustrating because we had to run the application over and over again.

Unit Test Coverage

Overall unit test coverage is 84% for package:main (which is all game logic and classes outside of the ui), but the ui's are not really tested

I think this is pretty good, while there are some smaller code branches not tested, the main missing parts are no code coverage of quite elaborate toString() override methods. These are long by # of lines and so are bringing the total down. I deem these not particular useful to test because they don't have any state changing effects, so the only real test I can do is a test to see if the string output matches, and do to that I'd have to remake an identical tostring() test method

Testing some probabilistic methods was hard. In my test setup pirates have a ⅓ chance of taking all your money, and I wanted to test that they did, so instead of modifying code to override probability for the test (there was no easy way to set seed for this section in my design) I just had pirates board 7 times in a row. So probability pirates do take over = 1 - (⅔)^7 =94% so the test works 94% of the time

Project Thoughts & Feedback

[Ben] This project took up far more time than I expected, and it was definitely the first time I've done something this big end to end. I'd say overall this makes this class quite a heavy class time commitment wise, but I'm not sure there is an alternative its good learning
[Kvie] This is the biggest project that I have ever done and hence, I have been struggling in some parts during the project. I think the tutorial was a big help but it could be improved by giving some more examples in implementing methods such as adding an image, adding a scrollbar to JList in Swing, etc.

Introspective on what went well, what didn't and improvements for next time

[Ben] Our teamwork was very poor, we needed more communication and needed to front load work a bit more. I was always a bit worried about how we were tracking. Kvie, I believe, was more comfortable, until she wasn't. In the last ~week Kvie has put in substantial effort to get the GUI completed.

[Kvie] Overall, the project went quite well. Everything was completed before the due date. However, I think our team has very poor communications which resulted in improper workload division. I also believe that all my working effort was not being taken into account. There are lots of things that I need to improve next time, especially in communicating with my partner and dividing the workload properly.

Effort Spend

Ben - 100-150 hours; Kvie - 80-100 hours

Statement of agreed % contribution

Effort (Ben - 60%, Kvie - 40%)

Output
[Ben] I have been unhappy with the quality of the output and progress I felt Kvie was making in the mid section of the project. While Kvie did some initial work, I largely did all game logic and the CMD UI. Until about a week ago, I thought we'd be handing in a project without a GUI. However, Kvie has done a huge amount of work in the last week or so, talking to tutors, talking to me, making code etc etc. The Gui is pretty complete and its mostly her work

[Kvie] I've struggled in writing code and understanding the game logic so I need lots of help from the tutors and my partner. I put in some initial work in the command application but wasn't able to finish because I didn't really get the idea of how to do it, which could have been better if I discussed more with my partner. When Ben finished the command application, I got more ideas of how the game works and hence, I was able to do most of the GUI. I agree that Ben did more parts of the project than me, so the percentage of contribution is fair.