ECM2414 Software Development

50/50

Development Log

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Date/Time | Duration | Roles | | Signatures | |
| *Person 1* | *Person 2* | *Person 1* | *Person 2* |
| 20/10 | 2h | Driver | Observer |  |  |
| 21/10 | 2h | Driver | Observer |  |  |
| 23/10 | 3h | Observer | Driver |  |  |
| 24/10 | 3h | Observer | Driver |  |  |
| 26/10 | 2h | Driver | Observer |  |  |
| 27/10 | 4h | Observer | Driver |  |  |
| 28/10 | 4h | Driver | Observer |  |  |
| 29/10 | 3h | Observer | Driver |  |  |
| 03/11 | 5h | Driver | Observer |  |  |
| 04/11 | 3h | Driver | Observer |  |  |
| 07/11 | 3h | Observer | Driver |  |  |
| 08/11 | 3h | Observer | Driver |  |  |
| 11/11 | 5h | Driver | Observer |  |  |
| 15/11 | 2h | Driver | Observer |  |  |
| 16/11 | 4h | Driver | Observer |  |  |
| 17/11 | 5h | Observer | Driver |  |  |
| 18/11 | 7h | Observer | Driver |  |  |

Code and Performance

# Classes

In terms of classes, alongside the required Card and Player classes, we've also decided to implement a Game (mediator) and Deck classes. While the Card and Player classes could easily be represented as numbers, having them as classes makes the code much easier to read semantically, and the Player class in particular needs to have attached functions. In addition to this, the Deck class is also needed to hold functions relating to given Decks- this also adds to semantic meaning.

The Game class is used to represent a game session, and the reason this is separate from the main CardGame class is that not only does Game not have to deal with input, it could also allow multiple games to be run from the main CardGame class if need be in the future. It also makes it very easy for these classes to be exported as a Library because it's entirely separate from the main running class (in this case CardGame). The Game class is responsible for setting up the Player and Deck instances based on input (given as arguments, sourced from the CardGame class), as well as giving out the initial hands to Players and ensuring there's a clear and useable link between itself and the Player and Deck classes

# Behavioral Design Pattern - Mediator

As mentioned above, the Game (mediator) class interacts with every game object as well as all the Decks and Players. In the Game class, every Player and Deck is given references to itself. Conversely the Game is given a list of references to all the Player and Deck objects- this allows communication between all of them but using the Game as level of local encapsulation to ensure each Player only has the minimum required access to other Players and Decks via itself.

One of the most important roles of the Game is to allow any given player to communicate with the rest of the players to tell them when they've won, e.g. Player 1 wins, so in order to interrupt the threads of the other players, Player 1 tells the Game that they've won, and in turn the Game tells all other Players that there is now a winner, subsequently interrupting their threads.

The second most important role of the Game is to manage when a player performs the atomic action of giving/taking a card. The reason this function is in the Game is because the act of either giving or taking a card involves multiple objects such as the GameLogger, therefore it helps communication between them. The use of the synchronized keyword ensures that no two players call the functions at the same time to allow for Thread Safety.

# Logging

Due to how much logging needs to be done, we've given it its own class called GameLogger. The way this works is that it's allowed to interact with all the Player and Deck classes, but only through the Game (mediator) class (due to our choice of Behavioral design pattern) via all the methods that are shared by Players such as discardCard and drawCard- both of these methods are always called by Players but send output logs to the GameLogger class. Throughout the game, lists of each Player's output strings are collected, then at the end after the final output function finishes, these lists are written to their appropriate files.

# Performance

We’ve decided that in order to take load off the individual player threads, the best approach for logging player moves is at the end of execution. Therefore, our GameLogger class contains methods corresponding to game actions that the Game class (mediator) can interface with and put onto a buffer which is written to file at the end of our program’s execution.

Testing with Junit 4.4

# Interfaces

To allow for easier mocking of our objects while testing, all objects that were to be mocked are not directly referenced in the main program but rather by an interface that they share with the actual mock classes. This allows for minimum changes to the production code for the sake of testing and makes the code more expandable for the future.

# Tests

Instead of referencing by interface in the case of the constructor for the game class, we’ve decided to add getters and setters to the game function to allow for easier access to private variables. This could also be done by reflection, however some sacrifices in performance and code readability meant that we took this other approach instead.