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, Deck, and Mediator 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 the Mediator and the Player and Deck

# Behavioral Design Pattern - Mediator

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

One of the most important roles of the Mediator 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 Mediator that they've won, and in turn the Mediator tells all other Players that there is now a winner, subsequently interrupting their threads.

The second most important role of the Mediator is to manage when a player performs the atomic action of giving/taking a card. The reason this function is in the Mediator is because the act of either giving or taking a card involves multiple objects, 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 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

Testing

# Interfaces

# Tests