Weiting Li, Nimesh Nayaju
COMP 3721

The implementation of our RobotTurtle game is structured in the following way:

Model
```
   |______ Board
   |______ Card
   |______ Game
   |______ GameOperation
   |______ Player
   |______ IMove (Interface)
               |______ LeftTurn
               |______ RightTurn
               |______ ForwardMove
               |______ Bug
   |______ BasicTile (Abstract)
               |______ Jewel
               |______ MovableTile (Abstract)
                          |______ Turtle
   |______ GameState
   |______ Direction
   |______ Position
   |______ TileInfo
```

Controller
```
     |______ RobotTurtleController
     |______ LogicController
     |______ DisplayFormat
     |______ ManipulateModel
     |______ PromptController
```

View
```
   |______ GameDisplay
```

Addition of Card class and *IMove* interface (Model)
We decided to get rid of the `Card` enum and create a new concrete `Card` class because of the change in our design decision explained later in the paragraph. We used the Dependency Injection technique via Constructor Injection so that a `Card` constructor always takes an `IMove` object. This is to say that a `Card` must always have a move (e.g. Forward, LeftTurn, etc) associated with it.
We also decided to have all the moves implement the `IMove` interface with an `execute(MovableTile tile)` method. This is to ensure if any new move is to be added (e.g.. ShootLaser), it must have an `execute()` method and in that way, we're keeping our design open to additions without modifying any codes already present. This is also to ensure we're adhering to the Dependency Inversion principle because we're adding a reference to `IMove` and not the classes extending `IMove`.

Addition of *BasicTile* and *MovableTile* abstract classes (Model)
A `BasicTile` in our design represents any time with the basic `Position` attribute, but a `MovableTile` which extends `BasicTile` has an added `Direction` attribute to determine where it can be moved. This is to adhere to the Interface Segregation Principle so that each interface has a single responsibility and to Dependency Inversion Principle because we can reference the interface without needing to depend on the low level module. Also, any new additions of tiles (Crate, StoneWall) would require implementing either `BasicTile` or `MovableTile` without any modification of the code, therefore adhering to Open Closed Principle.

Addition of ManipulateModel, DisplayFormat, LogicController, and PromptController classes (Controller)
We delegated the responsibility of converting the model information (turtle positions, directions) to the `ManipulateModel` class. It requests tile information from the `Game` class and massages them to a format understandable by the View - we created the `DisplayFormat` class in the controller to store this manipulated information. This also corrected our older design of using a `toString()` method on certain model objects to determine how to display them.
Furthermore, we've introduced a `LogicController` that handles the controller logic while `PromptController` deals with all the prompts that the controller sends to the user.

Updated View
We updated our view to receive information about tile positions (under DisplayFormat) from the ManipulateModel controller and pretty print them in the terminal.

Separated RobotTurtleGame into Game and GameOperation
To adhere to the SRP, the original `RobortTurtleGame` class is separated into two classes: `Game` and `GameOperation`. `Game` is responsible for getting/setting/storing the data, and

`GameOpeartion` is in charge of executing every step as the game goes. To illustrate, imagine `Game` as the library; all the information in the game are books stored in the library’ s collection. `GameOperation` is the librarian that has access and can manipulate these information: By assigning the turn, making the move, checking the validity of the cards and so on, `GameOperation` fetch all the data it needs from `Game` and then executes corresponding methods as the game progresses. They conduct two different tasks. According to SRP, it is necessary to put them in separate classes.