Controller

The controller is the class that represents the functionality of a Controller in the MVC paradigm. It is responsible for starting the game, and instantiating the two central view and model objects (those being the GameView and Board respectively). We choose to make most of the members of this class static. Because the controller isn't an actual thing in the game, but rather a meta structure for communication between the model and the view, it didn't make sense to instantiate it. Also, since every game should only ever have one controller, everything inside a controller can just belong to the class itself while maintaining full functionality.

The main responsibility of the Controller is the communication of events and state information between the View and the Model. In order to fulfil this responsibility the board contains a reference to the GameView and the Board. The stack of method calls for when a card is clicked is the following:

"CardStackListener.ActionPerformed() -> Controller.onCardChosen() -> Board.playTurn() -> TurtleMaster.onCardPlayed()"

Because there are a number of ways that a move can be deemed invalid, and the responsibility of deeming a move invalid falls across multiple classes, this stack of boolean function calls serves as an elegant way to signal back to the previous calling class that a move was not valid (which eventually will fall back to the view). If at any point, one of these methods returns false, the previously called method will also return false. This will then eventually fallback to the view which notifies it to prompt the player to pick a different card. Doing so will then start the call stack over again.

Controller Enumerations

The Controller package contains two different enumerations which represent card types and tile types. These are used as a common way to pass information between the View and the Model. We choose to use enumerations because of their flexibility and extensibility. For example, if a new TileType or CardType needs to be added, adding it to the enumeration will account for it in every section of the code. For example, the BoardPanel (responsible for drawing the board layout) has an array of tile sprites which is sized based on the number of elements in the TileType enumeration. This array is also indexed using the ordinal value of an element in the TileType enumeration. This was a design pattern we used for the Tile and CardTypes.

View.BoardPanel

The BoardPanel is responsible for drawing and updating the board layout. In order to keep this as separate and independent of the model as possible, it is able to track and update player position and direction based solely on the player number and new position. This update occurs every time a valid move is made. If an invalid move is played, the aforementioned method call stack terminates before the Controller instructs the view to update. The majority of the code in this class consists of graphics documentations

View.CardStack

The CardStack is an extension of a JButton. When designing this project, we knew we needed an easy way to translate a button click to a "type of card" without having to include model types in the view. This is originally what inspired the use of enumeration types. Each CardStack has a CardType, which is sent down the method call stack when a card is played. This type is then used by Model classes to determine how their state should update when a card is played.

View.CardChooserDialog

When designing the game, we decided to put each players “hand of cards” into a separate window. We choose to do this because it models the real world idea of each player having a set of cards which is detached from the game board. It also improved the simplicity of the UI design for the window which displays the board. The responsibility of the CardChooserDialog is to display the options that each player has during their turn, and then communicate the players decisions to the model. Originally, we had planned to keep track of each players remaining cards but decided to give a player unlimited moves for simplicity. It also seemed like there were always a sufficient number of cards for a player to get to the centre of the board and we didn’t feel like limiting card choice presented any challenge to the player in the current iteration of the game.

The CardChooserDialog has an inner class for CardStackListener. The CardStackListener. The CardStackListener is a custom action listener which begins the method call stack when a player chooses to play a card.

View.CardChooserDialog

When designing the game, we decided to put each players “hand of cards” into a separate window. We choose to do this because it models the real world idea of each player having a set of cards which is detached from the game board. It also improved the simplicity of the UI design for the window which displays the board. The responsibility of the CardChooserDialog is to display the options that each player has during their turn, and then communicate the players decisions to the model. Originally, we had planned to keep track of each players remaining cards but decided to give a player unlimited moves for simplicity. It also seemed like there were always a sufficient number of cards for a player to get to the centre of the board and we didn’t feel like limiting card choice presented any challenge to the player in the current iteration of the game.

The CardChooserDialog has an inner class for CardStackListener. The CardStackListener is an action listener which begins the method call stack when a player chooses to play a card.

Model.Board

The Board class is the central class in the model which is responsible for storing and updating all other game objects. The position of a game object is a single integer ranging from 0 to 64 (or whatever board size the game is to be played on). We choose to use a single integer as opposed to x and y values because it simplified the communication of a game object between classes. Because every position is a single integer, we choose to store the tile layout as a Hashmap<Integer, TileType> (with the integer key representing a position on the board). This decision made it very simple to quickly determine if a player is occupying the same space as a tile. For simplicity, any position which does not have an entry in the hash map is considered a blank tile. This was an easy decision to implement using the getOrDefault() method of HashMap.

Another design challenge was detecting collisions between turtles. We decided it would make sense for the board to store an array of occupied position, which can then be updated as the players move around the board. This is also extensible to add tiles which cannot be moved through in the future.

Model.TurtleMaster

The TurtleMaster class is responsible for determining how a Turtles position and direction should be updated when a card is played, and also as a front end for accessing information about the turtle it controls. It also is responsible for storing the if a player has won.

Model.Turtle

The Turtle class represents the turtle on the Board. It is responsible for updating its position when a card is played as well as checking that a given move is valid within the rules of the game.