**Panels**

There are three views corresponding to the Tile objects, CornerPanels, PropertyPanels and StreetPanels. Each of these extends JPanel, has a limited number of data members (String name, int width, height angle, and the array list of guests), which are passed as arguments to the constructor by the Tile to which each Panel corresponds. Each Tile, upon instantiation, in turn instantiates a corresponding Panel, to which it passes itself as an argument, allowing the Panel to set its data members in accordance with the Tile’s properties.

Because the Panels must be rotated as they wrap around the outer edge of the game board, they must have an assigned rotational angle, based on their position on the board. This angle is calculated from the corresponding Tile’s “number.” Each row is made up of ten tiles. If a Tile has a number of 14, it must be on the left side of the board and thus must be rotated 90 degrees.

The paintComponent of each Panel rotates and translates the coordinates, draws rectangles or appropriate size and orientation, draws the icon of each guest occupying the Panel’s Tile and draws the name of the Tile. The placement of the Tile’s name differs depending on the type of Panel and StreetPanels additionally have smaller, colored rectangles in their upper quadrants. The variation between these types of Panels is slight, thus, a strong case could be made for a restructuring of the code to include a class hierarchy, such that one TilePanel would include all of the repeated paintComponent methods and the Corner-, Property-, and StreetPanels would extend this class, including an additional paintComponent, which would call its parent paintComponent, and then draw its name and colored box where appropriate. Again, due to time limitations the parent TilePanel was not implemented. I managed to collapse the more numerous types of Panels I’d originally created into the three classes used here, but the addition of the TilePanel introduced bugs that I could not fix quickly enough.

*setUpBoard*

SetUpBoard is a static method found in the Monopoly.java, a file which is generally responsible for the higher level setup of the game board and thus serves as something between a model and view for Tiles. After defining some custom Colors, setUpBoard goes through the tedious process of instantiating each Tile, individually, which unfortunately can’t be expedited, as noted above, because the properties of the Streets in particular follow no pattern and must be hard coded. Each Tile is added to an array list, which the Player will reference to get his or her position on the board. This array list is then loaded in four pieces to four larger JPanels. Each of those four panels is loaded into a larger gameboard JPanel, which is the component that will directly be loaded into the JFrame.

**Players and Their View**

The final major classes are the Player and ControlPanel. In a not entirely obvious way, the ControlPanel could be seen as the view (though it’s a controller as well) of the Player model/controller.

* *Player*

The Player class, as the name suggests, contains the methods most obviously associated with the actions a person playing Monopoly would take. The Player has a number of boolean switches that determine actions he or she can take during the turn and whose values are set by beginTurn and endTurn functions (called by the ControlPanel), running (int) totals of funds and non-cash assets (properties, houses and hotels), and an array list of deeds, which collects the properties that the Player owns.

The next few methods deal with Dice, a helper class that has a simple diceRoll method (spits out two random numbers from one to six), which passes a roll value back to the Player and calls Player’s doubleRoll method, and a method for setting an image (Dice is also a JPanel) corresponding to the dice roll.

On his or her turn, a Player can roll dice only once, thus the diceRoll method initially sets canRoll to false. If the result of that roll is doubles (each die shows the same number), canRoll is set back to true, allowing the Player (in fact forcing the Player) to roll again. The doubleRoll method keeps track of doubles per turn, since three double rolls sends the Player to jail.

If the Player has not rolled three doubles and is not in jail already, the result of the roll is passed to the Player’s move method. Move calls the current Tile’s leave method, then the landedOn method of the Tile “rollResult” number of spaces higher in the tileList. A second move method is called when, rather than moving a dice roll’s number of spaces forward, the Player is sent directly to a particular Tile. In my implementation, the only time this happens is when the Player lands on the GoToJail Tile, however, the method is set up to accommodate the other cases, which would be caused by Community Chest and Chance card draws.

When a Player is sent to jail, he or she can then attempt to get out of jail by rolling doubles, or else pay a fine; in effect calling the getOutOfJail method, which decrements funds and reenables the Player to move in the next turn.

During his or her turn, a Player can also choose to upgrade a property, as described in the StreetPanel section. The Player’s upgrade method populates a list with potentially upgradeable properties, in this case, the contents of the Player’s deeds, an array list of Properties owned by the Player, and asks the Player to select a property from this list. This would call the corresponding Street’s upgrade method.

Finally, there’s a small number of setter methods, all dealing with purchases and rents and thus called by Properties, and a longer list of getter methods. These are used by either Tiles or the ControlPanel to access most of the Player’s data members.

* *ControlPanel*

The ControlPanel, (as suggested by my naming convention) an extension of JPanel, serves as both a view and a controller for the Player. As a view, it is fed the Player whose turn it is, and displays this player’s name and funds. As a controller, it also features a number of JButtons, which, when clicked, call various Player methods (rollDice, upgrade, getOutOfJail, and endTurn). This gives ControlPanel the marked distinction of being the only component with which the human player can directly interact (aside from the option panes thrown up by various other sources).

Because it was already responsible for managing each Player’s turn, ControlPanel could also manage the entire flow of the game. Thus, within ControlPanel is a takeATurn method, which takes as an argument a Player whose view and control is then managed by the ControlPanel. Once the human player presses the end turn button, so long as the Player has no additional rolls to make, the Player’s turn ends, and the next Player in a circular linked list (another utility class I built) is passed to the takeATurn method.

As each Player progresses around the board, he or she is bound to trigger various events that change the amount of his or her funds. Each time the Player’s payment method is called, the ControlPanel’s updateFunds method is called to display this new amount. If ever updateFunds discovers that a Player’s funds have dropped below zero, that Player is dropped from the game (i.e. he or she is dropped from the circular linked list). If only one Player remains in the list, that Player wins.

Additionally, the ControlPanel serves as a view for the Dice, though this was less due to any semantic logic than to convenience: namely, I didn’t want to go to the trouble of including yet another JPanel in the JFrame.

**Leftovers**

The only code I have yet to explain is all fairly straightforward. Monopoly’s main method calls setUpBoard, instantiates Players based on user input, instantiates the ControlPanel and starts the game by calling ControlPanel’s takeATurn method.

Mentioned above, the ControlPanel takes advantage of a CircularLinkedList (which is populated with the Players instantiated in main), a class very similar to the LinkedQueue discussed in class; its main difference being that the tail node always links to the head, so that calling getNext will eventually loop through the list. The CircularLinkedList in turn takes advantage of the Node class, which is almost identical to the QueueNode class.

Sample Game Play

In the initial set up the user indicated that there would be two players, named Player 1 and Player 2. To begin the game, the main method passed Player 1 to ControlPanel’s takeATurn method.

TakeATurn sets ControlPanel’s player to Player 1, sets a label to “Player 1,” calls updateFunds, which calls Player 1’s getFunds method. GetFunds returns 1500, which is then set as the text of another JLabel displayed in the ControlPanel. Returning from updateFunds, takeATurn finally calls Player 1’s beginTurn method.

BeginTurn sets Player 1’s myTurn to true, canRoll to true and then returns. Now we idle while ControlPanel listens for various button click events.

(Human) Player 1 presses the roll dice button, calling ControlPanel’s actionPerformed. This method determines that it was the roll dice button that was pressed, calls Player 1’s getCanRoll, which returns true, and thus calls Player 1’s rollDice. Player 1 canRoll and is not inJail, so canRoll is set to false and Dice’s roll method is called.

Roll generates two random numbers, a 2 and a 2. Because the two numbers are equal, Player 1’s doubleRoll is called.

DoubleRoll sets canRoll back to true, increments doubles (from zero to one) and returns.

Roll calls setImage twice, which calls setIcon on each die, changing both images to two.png. Finally, roll returns 4.

Back in Player 1’s rollDice, rollResult is set to 4, and as inJail is false, move is called and passed 4 as an argument.

Move sets an int to the currentTile’s (Go) number plus one less than the roll (since tileList begins at 0, not 1). Go’s leave method (inherited from Tile) is called, updating Go’s guests (by removing Player 1) and calling Go’s updateGraphics. UpdateGraphics calls Go’s Panel’s update method, removing Player 1 from its list of guests and then redrawing the Panel without Player 1.

Back in Player 1’s move, a destinationTile is set as the member of the tileList 3 positions away from Go (INCOME TAX – oh no!). Player 1’s currentTile is set to INCOME TAX and INCOME TAX’s landedOn method is called.

INCOME TAX’s guests list is updated to include Player 1, his icon in drawn by INCOME TAX’s Panel (via updateGraphics) and an option pane pops up asking if Player 1 wants to pay $200 or 10% of his total worth. (Human) Player 1, knowing how the game works, chooses the 10% option. INCOME TAX calls Player 1’s getTotalWorth, which returns 1500, multiplies this value by 0.1. Player 1’s payment method is called with an argument of -150. This method in turn calls ControlPanel’s updateFunds, which reprints the funds label to read, 1350 and then pops open an option pane congratulating him for not being stupid.

Player 1’s move having now finished, ControlPanel goes back to idly waiting for a button click. Player 1 presses end turn. ControlPanel calls Player 1’s getCanRoll method, which returns true. An option pane pops open, reminding the player that he must finish rolling. Player 1 presses the roll dice button.

The steps above are repeated, Dice’s roll returning a 3 and a 4. Move is called with 7 passed as an argument. INCOME TAX is left, and ST. JAMES PLACE is landedOn. ST. JAMES PLACE’s landedOn (inherited from Property) adds Player 1 to the guests list, updates the Panel, then checks to see if the Tile has an owner. It does not, so an option pane opens, asking if Player 1 would like to purchase this Tile. Player 1 presses the buy button. ST. JAMES PLACE calls Player 1’s getFunds, which returns a value larger than ST. JAMES PLACE’s price. Player 1’s payment and addAsset methods are called and passed the price, Player 1’s addDeed method is passed ST. JAMES PLACE, the Tile’s owner is set to Player 1 and an option pane congratulates him on the purchase.

Back to idling, Player 1 now presses end turn and since the last roll had set canRoll to false, this time ControlPanel calls Player 1’s endTurn method.

EndTurn asks player 1 if he’s sure, and he presses yes. It sets myTurn, canRoll, canPayFine and canUpgrade to false, sets doubles back to zero and returns true, so that the ControlPanel knows to now getNext player from the CircularLinkedList and pass Player 2 into its takeATurn method.