Michael Juhasz

CSCIE 10B Final Project Write-up

***Instructions***

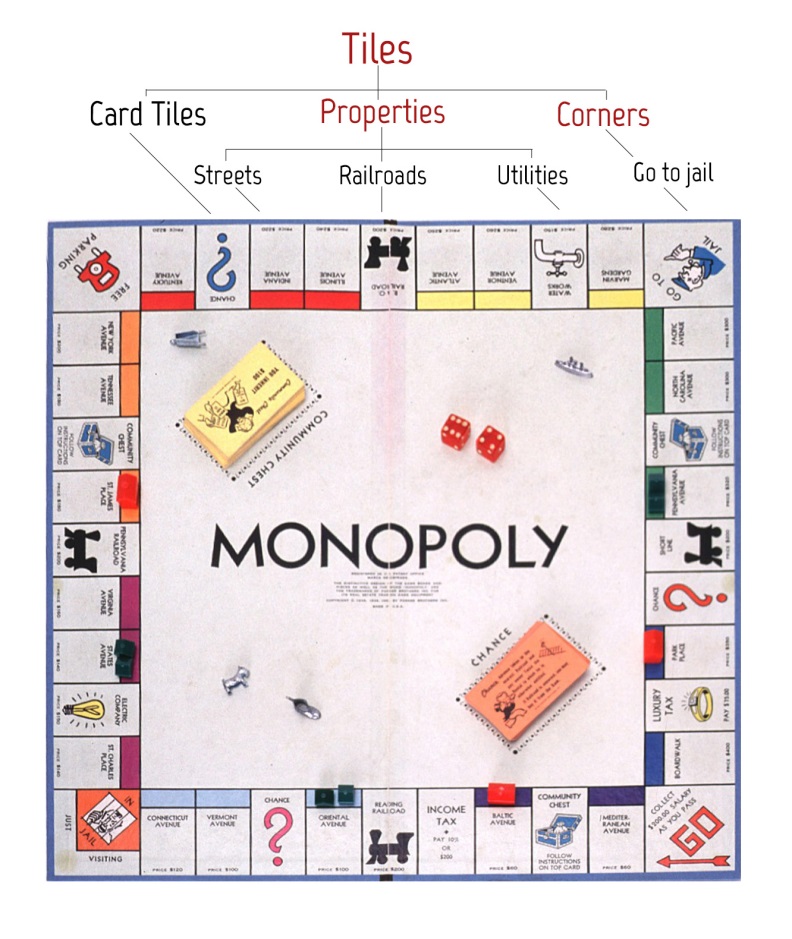
Run the Monopoly.java file. Follow the instructions presented by the option panes. The input is basically idiot proofed. You can choose as many or as few players as you want. Zero players doesn’t crash the program, though it does throw a runtime error – however, this feels like the correct functionality. If a user wants a zero player game, he’ll be able to look at the empty game board and that’s it.

Once the game has begun, the player whose turn it is should begin by pressing the roll dice button. Every other choice either inevitably follows, or else is directed by option pane messages.

***Description***

For my final project, I have chosen to build a Java version of the board game Monopoly. In a non-technical sense, the program works just like a game of Monopoly would, with a few minor exceptions. Due simply to a lack of time, not all of the gameplay mechanics have been implemented. “Chance” and “Community Chest” tiles have no cards associated with them (and in fact trigger no action) and players cannot mortgage, sell, or trade properties. I didn’t include these features because they were substantially different or more difficult than any of the other implemented features, but merely because of time limitations and because the program seemed quite long enough already. Those familiar with Monopoly, will recognize in my program a player’s ability to roll dice, move around the game board, land on different types of tiles, some of which can be purchased and upgraded, causing other players who land on these tiles to owe you rent.

More technically, in designing this project, I’ve tried to use a model-view-controller pattern (though not Swing’s built in MVC capabilities) and thus the bulk of the code can be broken into two main categories; one being a hierarchy of “Tile” classes (combinations of model and controller) and a series of nested Swing components (the views).



Let’s begin with the models/controllers. The Monopoly board, as shown in the diagram above, is largely (large in gameplay significance, if not in physical space) made up of rows of tiles (officially called “spaces,” which is too semantically lightweight for me, thus the name change). The tiles come in four main classes, card tiles, tax tiles, corner tiles and properties, the latter two having subclasses of their own. This class hierarchy that I have here lain out and visually diagramed above perfectly matches the programmatic class hierarchy in my code.

**Tile**

All tiles belong to the parent class Tile, which is abstract and contains the data members and methods common to all its children. Every tile has a number, a name and an array list that contains the Players (a class to be described below) currently on the tile, as well as methods that get called when a Player lands on the tile and leaves the tile. The landedOn method simply adds the Player that was passed as an argument to the guests array list and calls the (abstract) updateGraphics method, which each subclass implements to update their associated views. The leave method removes the player from the array list and calls updateGraphics.

* *CardTile*

In my implementation, the simplest subclass of Tile is the CardTile. As stated above, “Community Chest” and “Chance” tiles have not been implemented and thus, landing on either of these types of tile triggers no actions, other than the Tile’s landedOn method. Thus, the CardTile merely constructs a CardTile, setting its name and number and instantiates the view associated with it (a PropertyPanel).

* *Corner*

Corners, or rather, some Corners are slightly more complicated than CardTiles, because two of them have landedOn methods that override Tile’s.

- *Go*

The Go Corner’s landedOn method overrides Corner’s, essentially mirroring it, but additionally adding $200 to the Player’s funds.

- *Jail*

Though a Player who is inJail must stay on the Jail tile, Jail itself does nothing beyond the normal functionality of a Corner. It exists as a separate class both for consistency (so that Corner can remain abstract) and so that in a future extension of this project, a unique image could be placed on the tile.

- *FreeParking*

The same case as Jail.

- *GoToJail*

GoToJail does as the name promises, sending the Player who landed on it to Jail, using the Player’s goToJail method.

* *Property*

Properties, like corners, are abstract, but contain significantly more unique code, included additional data members (owner and price), a significantly more complicated landedOn method and an abstract payRent method. LandedOn checks to see if the Property has an owner. If not, the Player is asked if he or she would like to purchase the property. If the Player does, the Player’s funds are checked against the price of the Property; passing this check, the Player’s deeds, funds and assets are updated. If, on the other hand, the Property is owned and the owner is not the Player, the payRent method, which differs based on the Property type, is called.

- *Street*

Streets are by far the most complicated Tiles. They have very numerous data members, which differ unpredictably between one instance and another (you might think, for instance, that a pattern could be found for the amount that each additional upgrade raises the rent, but no such patter exists across all Streets), and thus must be passed as arguments to the constructor.

Streets come in groups (represented visually by a colored block in the top quadrant of the tile) and each Street’s group must be defined by the setGroup method after all of its members have been instantiated. Two setGroup methods exist as some groups have two members, some three.

The rent to be paid by the non-owner Player who lands on a Street (thereby calling the payRent method) is determined by whether or not the owner has a monopoly (checked with the checkMonopoly method) on that group and by how many upgrades the Street has.

If a Player owns every Street in that Street’s group, he or she may, funds permitting, purchase upgrades for any of that group’s members. As explained in an inline comment, Streets must be upgraded equally; i.e. no Street can be upgraded if it already has received more upgrades than any other group member.

- *RailRoad*

RailRoads are similar to Streets, but cannot be upgraded. Their payRent method works similarly to Streets’, in that it’s toll is also based upon how many other like Properties the owner owns. In this case, however, because all RailRoads are of the type RailRoad, rather than taking an additional step to group them together, we can simply check to see how many RailRoads are included amongst the Player’s deeds.

- *Utility*

Very similar to RaidRoads. There are two utilities, they cannot be upgraded and their rent goes up if one Player owns both. As with RailRoads, Utilities’ rents go up if their owner owns both of them.

* *TaxTile*

TaxTiles, like Corners and CarTiles cannot be purchased. Their landedOn method checks to see which kind of the two TaxTypes (an enum type) the TaxTile is. If it’s LUXURY, the Player must simply pay $75. Otherwise, the Player is asked if he or she wants to pay $200 or 10% of his or her total worth. In the Monopoly board game, the rule is, you’re not allowed to calculate your total worth, which is the sum of your cash and the value of all your properties, houses and hotels, until after you’ve made this decision. Thus, in my version of the game, while the Player class keeps track of nonCashAssets and funds and will total them when its getTotalWorth function is called, this value is only ever shown after a player chooses the 10% option.

**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.

***Expansion***

As mentioned above, this implementation of Monopoly is ripe and ready for expansion. Community Chest and Chance tiles should have a landedOn method that grabs a Card from some list, each of which causes something to happen to the Player; mostly either sending the Player to some Tile, or calling payment with various arguments.

The layout could be improved, getting the corners to line up a bit better and making the text display more nicely. Icons should probably be drawn on each of the Corners and on the RailRoads and Utilities.

Most significantly - from a gameplay perspective (none of these changes requires major alterations of the existing code, since I wrote it with these expansions in mind) – Players should be able to mortgage and trade their Properties. Mortgaging a property would call the Player’s payment method, passing it half of the Property’s price, disable the payRent method, and allow the Player to repurchase the Property (I suppose in a fashion similar to how Property upgrades are selected). Trading would allow a Player to offer to transfer ownership of Properties, which would really be as simple as switching the Properties’ owners and updating the Players’ deeds lists. Both of these actions would probably necessitate another button or two on the ControlPanel.

There are additional gameplay expansions that could be implemented, though they tend to deal with aspects of Monopoly that only the most serious players encounter. Before attempting this project, for instance, I had never considered that in the board game, the bank has a limited number of houses and hotels, and thus buying up as many houses as possible and not upgrading to hotels is a reasonable strategy. Similarly, players can sell houses, effectively downgrading them, which opens up even more complicated scenarios. For instance, if a player downgrades from a hotel to four houses, but there are only three houses left in the bank, and the other two properties in the group also have hotels, the first property would have to be downgraded to three houses (since there aren’t enough for four). Then, because properties must be upgraded equally, and a hotel is more than one step away from three houses, the other two properties would have to be downgraded to three houses. However, there aren’t nine houses left, there are only three, thus all three properties would end up with one house each.

This situation would practically be easier to program than to explain. The only thing required is a downgrade function with a set of checks to the other group members’ upgrades, calling their own downgrade methods, if need be.

***Reactions***

I’m not sure if I think undertaking this project was a great idea. I absolutely learned a lot. My understanding of class hierarchy, inheritance and structures like linked lists, was cemented, and though I realize I still have much to learn about Swing layouts, I now know much more than I did a few weeks ago. This also served to greatly enhance my understanding of the MVC pattern, which I have only recently begun to learn about.

The only reason I’m uncertain about this project’s advisability is simply because Monopoly has a lot of pieces. Figuring out how to properly draw a Street Panel is a good lesson. Also figuring out how to properly display three other kinds of tiles is less interesting. To really see what I’m talking about, look at the setUpBoard method. It’s almost 200 lines long and all it’s doing is loading an array list, repeatedly calling a setter function and adding panels to other panels. In short, this project involved a lot of tedious work, which I had to spend time doing, rather than further expanding, or improving on the interesting parts of the code. Also, the length of the program meant that writing the code comments is probably a more onerous task than it was meant to be.