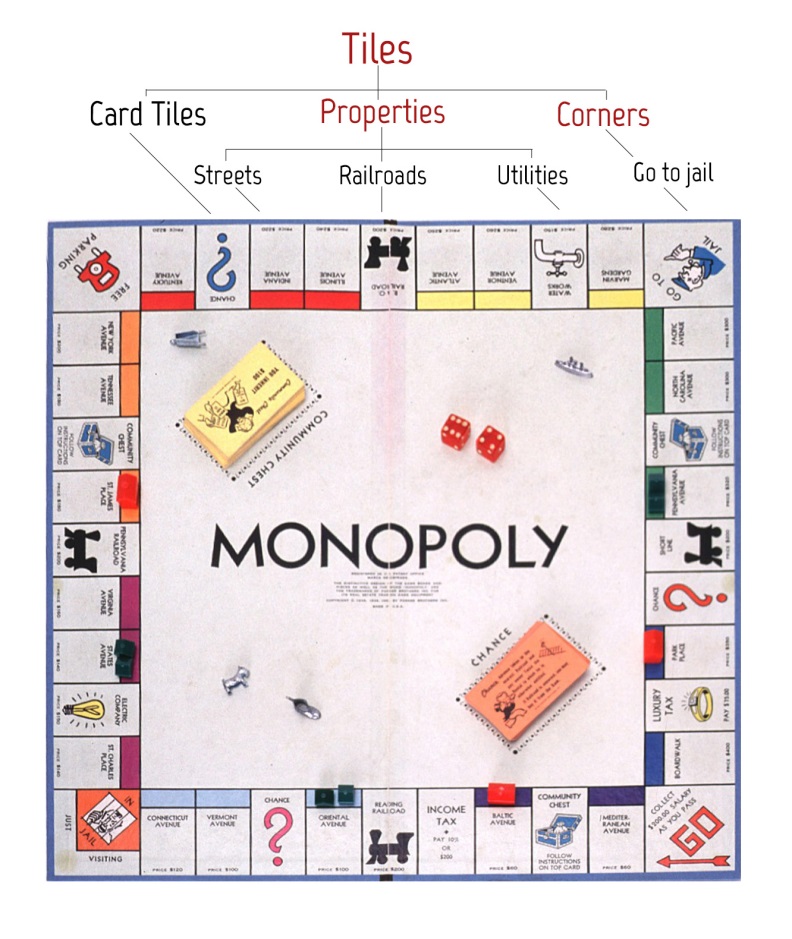
Michael Juhasz

CSCIE 10B Final Project Write-up

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 three main classes, card 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