Final Project Report for ECE 309

Matthew Denoncourt, Ilena Johnson, Justin Dhillion, Ethan Wilson

Link to Github Repository: https://github.ncsu.edu/jtuck/ECE-309-Monopoly

Rule Simplifications and Additional Features

We chose to implement monopoly for our project. However, we had to employ the following rule simplifications.

- There is no trading or bargaining between players in our implementation

- When the player gets out of jail, they stay on the jail space until their next turn.

- Players are not required to buy/hold the same number of houses on each space of a monopoly. When the player has a monopoly, houses and hotels can be bought individually for each property.

Additionally, because there are over a hundred cards with data that needed to be instantiated as our objects, we wrote a program to take the data from a TSV (tab separated value) file and instantiate all of our card and tile objects.

Overall Game Design and Implementation

R1.

To implement the game, we created a class for every physical object in the game. There are different base classes for deck cards (chance and community chest) and ownable cards (properties, rail roads and utilities) that inherit from a common "Card" base class, which only stores the card's name, identifier, and type. We have the following classes derived from these.

From OwnableCard, which stores mortgage and sale values.

- **RailRoadCard:** Stores rent value and a pointer to its associated rail road tile.

- **PropertyCard:** Stores rent value, house cost, hotel cost, and a pointer to its associated property tile.

- **UtilityCard:** Stores rent value and a pointer to its associated utility tile.

Each of these derived classes implements three virtual functions from OwnableCard, which mortgage, unmortgage and change ownership of their linked tiles through the pointer. They also add a pointer to the card to a map variable within the player class. These need to be virtual because the type of the linked tile pointer is different for each class.

From DeckCard, which only stores the card's description and contains a virtual function for its associated action.

- **getMoneyCard:** Gives the player money from the bank or takes money from the player.

- **transferMoneyCard:** Transfers money between two or more players.

- **getOutOfJailCard:** Can be held by the player until used to get out of jail.

- **movePlayerCard:** Moves the player to a different space while, depending on the card, checking to see if they passed go.

All of these actions are implemented through the virtual function doDeckCardFunction(), which can be called from pointers to a DeckCard. Calling all of these actions from a common pointer type allows us to store the cards in a STL deque object for chance and community chest without having to type cast.

We have similar logic for our type tile classes, which represent spaces on the board. We have an abstract base class called Tile that contains the virtual function doTileFunction(Player * P), and a derived class for each type of tile that implements this function to perform whatever actions should be taken when a player lands on the tile. We designed the classes this way so we can store the board as an array of Tile pointers and use the player location to access array elements. When a player lands on a tile, we can simply call doTileFunction(Player * P), without knowing what type of tile they landed on.

To support our classes, we have a series of functions defined in a separate header that perform common actions such as buying proeprties, checking the player's balance to make sure they can pay rent or buy properties, displaying properties for the player, and removing player's from the game when they go bankrupt.

Having these functions allows us to use a relatively simple main function that mainly calls functions and class methods.

R2.

The players themselves are stored as a separate class called Player, which contains member variables such as name, location and whether they are in jail. It also contains three STL unordered_map variables that store pointers to the property cards the player owns, their mortgaged properties, and their "Get out of Jail" cards (the only deck card that can be owned) respectively.

The computer player is implemented using the same class as the console player but employs different functions to interact with the game logic. During the player's turn, the main game loop checks if the player is a computer and calls different functions. These functions run mostly identical code but without the console input required from a human player. The computer player follows a simple set of rules, such as always buying properties when it lands on them and not voluntarily spending more than a certain amount of its balance buying properties, that we found to be optimal without excessive complexity.

R3.

Throughout the game, all of the player and game options are printed onto the console. This allows the players, or an observer if all players are computers, to watch the game and understand all of the logic that is happening as players of the physical board game would.

Project Requirements

C1.

Encapsulation: Our project demonstrates encapsulation in that almost all of the game logic and actions taken in association with a given card, space or player are performed by methods in our classes. For example, if a player lands on a property tile owned by another player, we would make the following method call Board[P->location]->doTileFunction(Player * P);. The method would then calculate the rent, checking if the property is a monopoly or has houses, check the paying player's balance, and transfer the rent money.

Abstraction: We had some difficulty with this concept because, in a board game like monopoly, almost all of the game's information should be public. Other players and the game should be able to see a player's balance, location, owned and mortgaged properties, etc. Further, most of the tile and card actions need direct access to this information to work properly and it didn't make sense to write methods that do nothing but pass this information. However, there is one thing in the monopoly game that should not be visible to other objects, the contents of the chance and community chest decks. Given this, we implemented these decks as private members of their tile and have methods to enqueue and dequeue the cards without making the entire deck visible.

Additionally, we used the STL whenever possible throughout this project, objects of which abstract their member contents by default. For example, when looking for a card in a player's deck of owned properties, we use ownedProperties.find().

Inheritance: As described in our overall game design, we have many derived classes that inherit and use members and methods of our base classes.

Polymorphism: Several of our base classes, such as DeckCard, OwnableCard and Tile are abstract base classes that contain virtual functions. As described above, these are essential to our code and properly represent the game's intended logic.

C2.

We used the STL for most of the objects contained within classes in our code, which employs good design as a rule. Additionally, because we only have one instance of each object, all of our class members that are not STL are either pointers to other objects or simple data types. We never need to copy, only point to, these objects so this implementation works and does not create memory leaks. All of our objects declared on the heap on instantiation should exist for the entirety of the game and are deallocated when main concludes.

C3.

We have an abstract base class called Tile that contains the virtual function doTileFunction(Player * P), and a derived class for each type of tile that implements this function to perform whatever actions should be taken when a player lands on the tile. We designed the classes this way so we can store the board as an array of Tile pointers and use the player location to access array elements. When a player lands on a tile, we can simply call doTileFunction(Player * P), without knowing what type of tile they landed on.

C4.

One of the data structures we used was a STL unordered_map to store the player's owned cards, mortgaged cards, and get out of jail cards. We used this because we know that it is implemented as a hash table and supports O(1) lookup and insertion time. This is important because we need to check these decks for certain cards and insert cards into them frequently throughout the game. We almost never need to iterate over all the cards and, by the nature of the game, the order of these decks does not matter.

Team Member Contributions

Matthew:

- Took lead on planning our game logic and how to translate game mechanics into code
- Wrote the original player, tile, and card base classes
- Wrote the classes derived from DeckCard
- Worked extensively on debugging our final code

Ilena:

- Wrote the program that handles our file input and object instantiation
- Developed the computer player functions based on our console player functions
- Worked extensively on debugging our final code

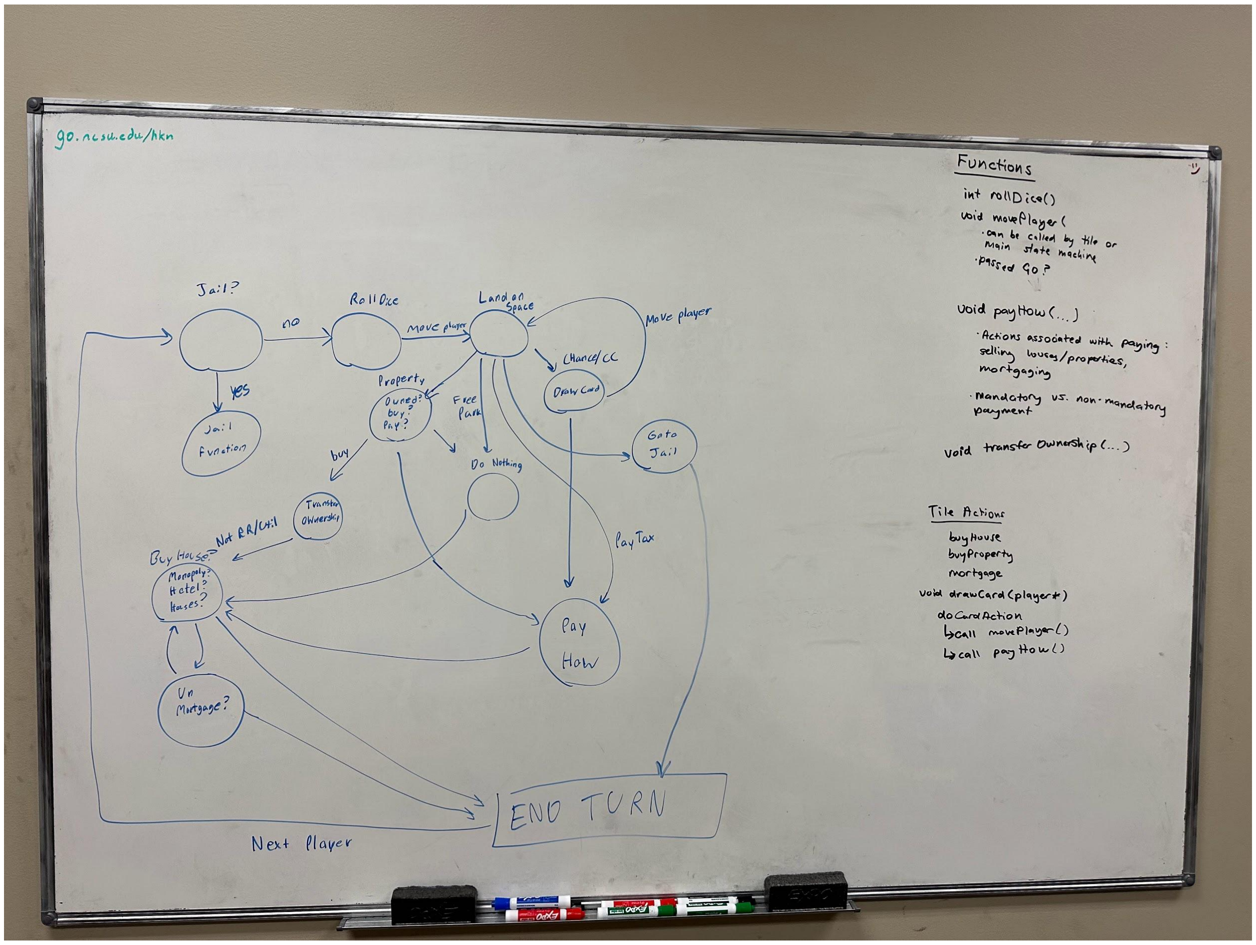
Justin:

- Designed layout for mid game testing (See Figure 2)
- Wrote pseudo-code for main game loop
- Implemented shuffling cards function, diceRoll function, including rolling doubles. go to jail after doubles 3 times

Ethan:

- Wrote all of the classes and methods derived from Tile and OwnableCard
- Modified the player object to include STL unordered_maps for card decks
- Wrote most of the functions in functions.h and function methods in functions.cpp

Images:



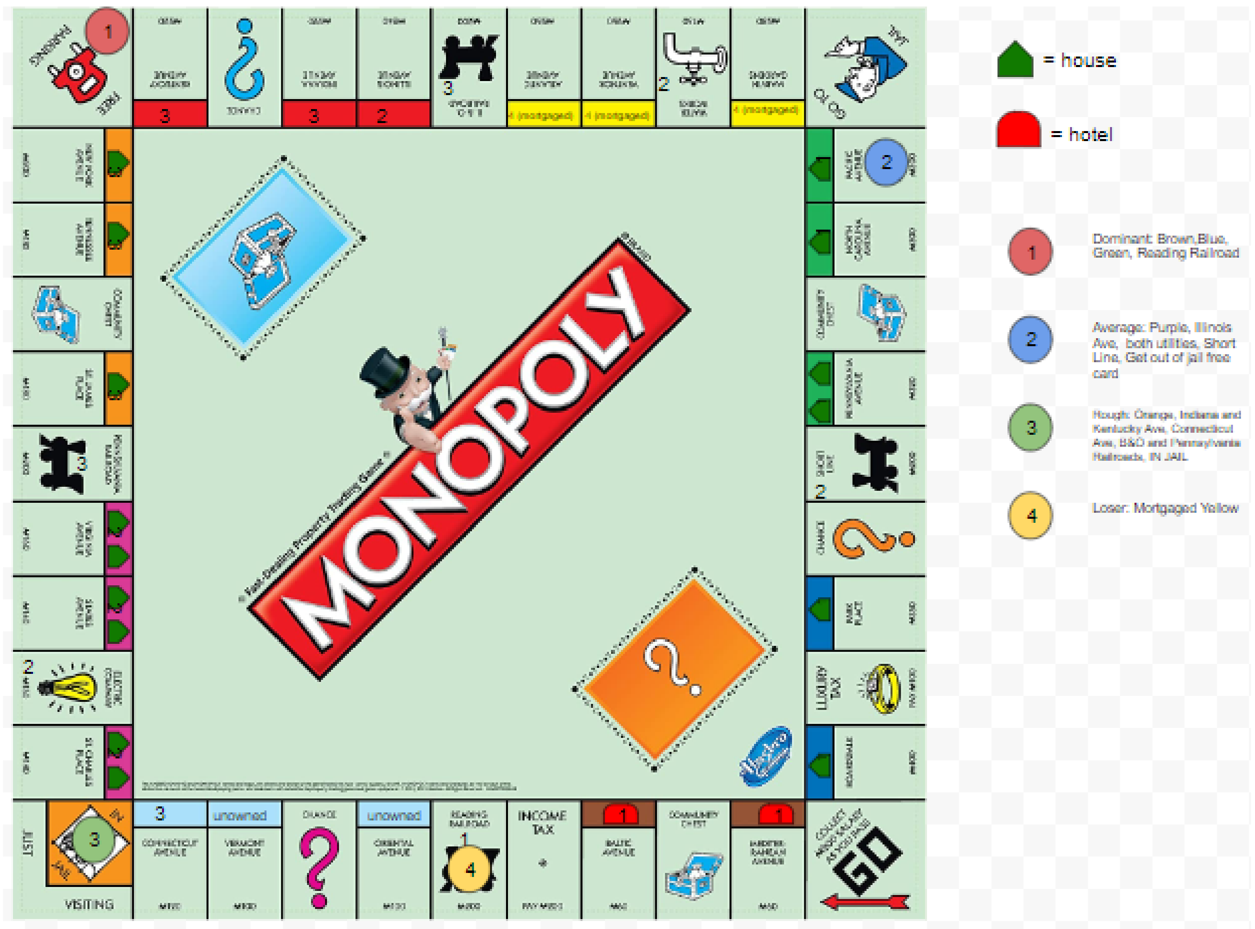
```
Functions
  int rollDice()
  void movePlayer(
      ·can be called by tile or
       Main state machine
      ·passed Go?

  void payHow(...)
      ·Actions associated with paying:
       selling houses/properties,
       mortgaging
      ·Mandatory vs. non-mandatory
       payment

  void transferOwnership(...)

Tile Actions
    buyHouse
    buyProperty
    mortgage
  void drawCard(player*)
    doCardAction
     ↳call movePlayer()
     ↳call payHow()
```

Figure 1



= house

= hotel

1 Dominant: Brown,Blue, Green, Reading Railroad

2 Average: Purple, Illinois Ave, both utilities, Short Line, Get out of jail free card

3 Rough: Orange, Indiana and Kentucky Ave, Connecticut Ave, B&O and Pennsylvania Railroads, IN JAIL

4 Loser: Mortgaged Yellow

Figure 2