Monopoly Game

Project Report

Group 8

Poe Il Park

Aigerim Mukusheva

Josh Fijal

Problem Statement

Most casual games of the Monopoly board game end up requiring several hours to finish based on the classic Monopoly rules, and the game loses most of its entertainment value and evolves into an achingly long trial of human endurance. The players may no longer find the game "fun" but still feel too competitive to simply forfeit to the richest participant, thus prolonging an exercise in drudgery that was ironically meant to be lighthearted amusement.

Though our original proposal involved creating a Monopoly program identical to the classic board game, due to an approaching deadline for the project and a lack of time to implement all the features in the proposal, it was decided that a simplified version would function effectively as a computer game. It was also reasoned that this version may ultimately prove more tolerable for the average user than if all features proposed had been implemented by making the game less complex and typically each game shorter in duration. In particular, the endgame condition of bankruptcy without the option of mortgage ensures an average game length shorter than that of the classic board game.

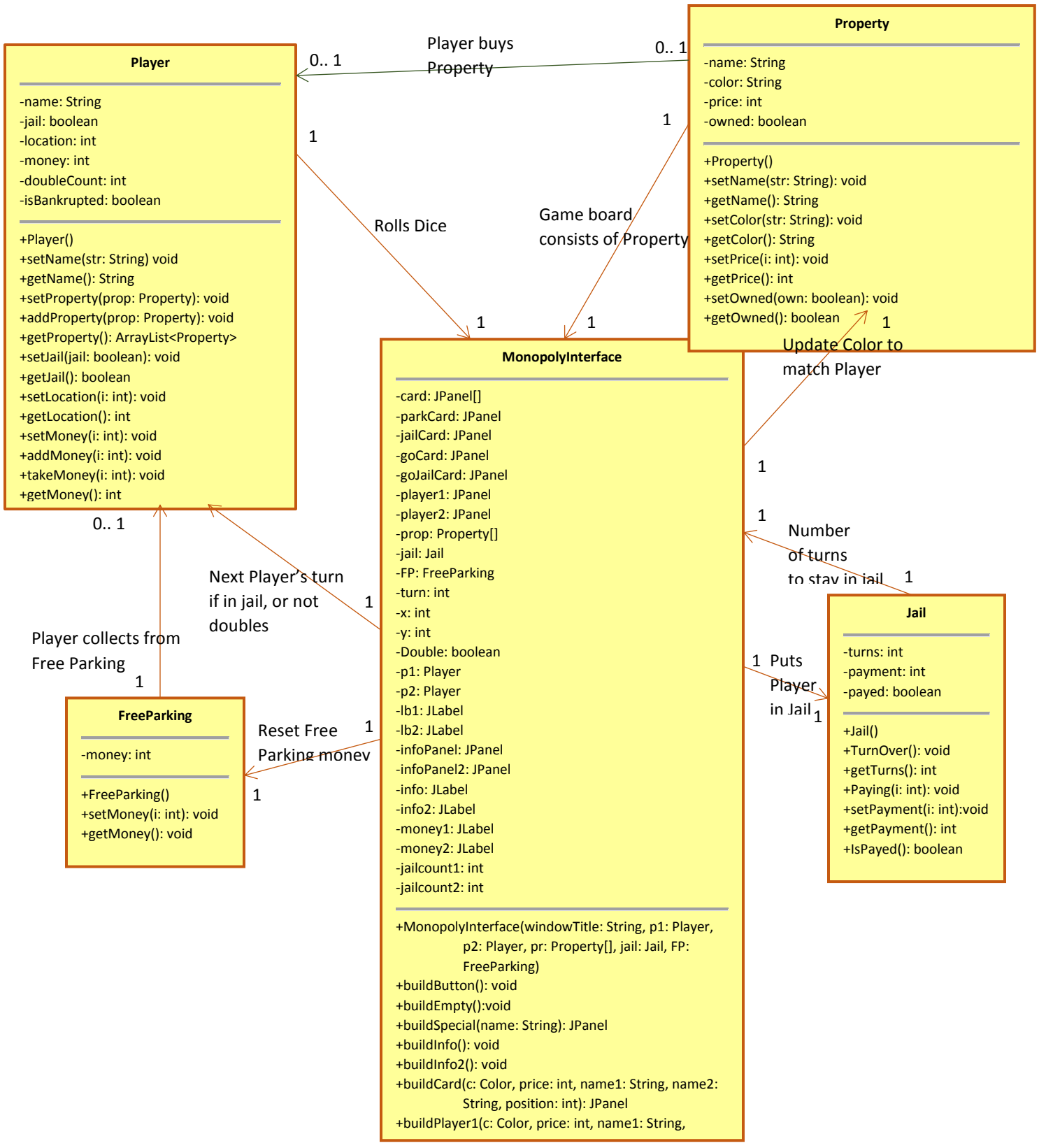
Another significant change in the rules of this version of Monopoly that reduces its complexity is the elimination of the bartering aspect of Monopoly. However, coupled with the lack of houses and hotels, this omission should not detract from the entertainment value of the game, as there is really no incentive to attain a monopoly of properties.

System Boundary

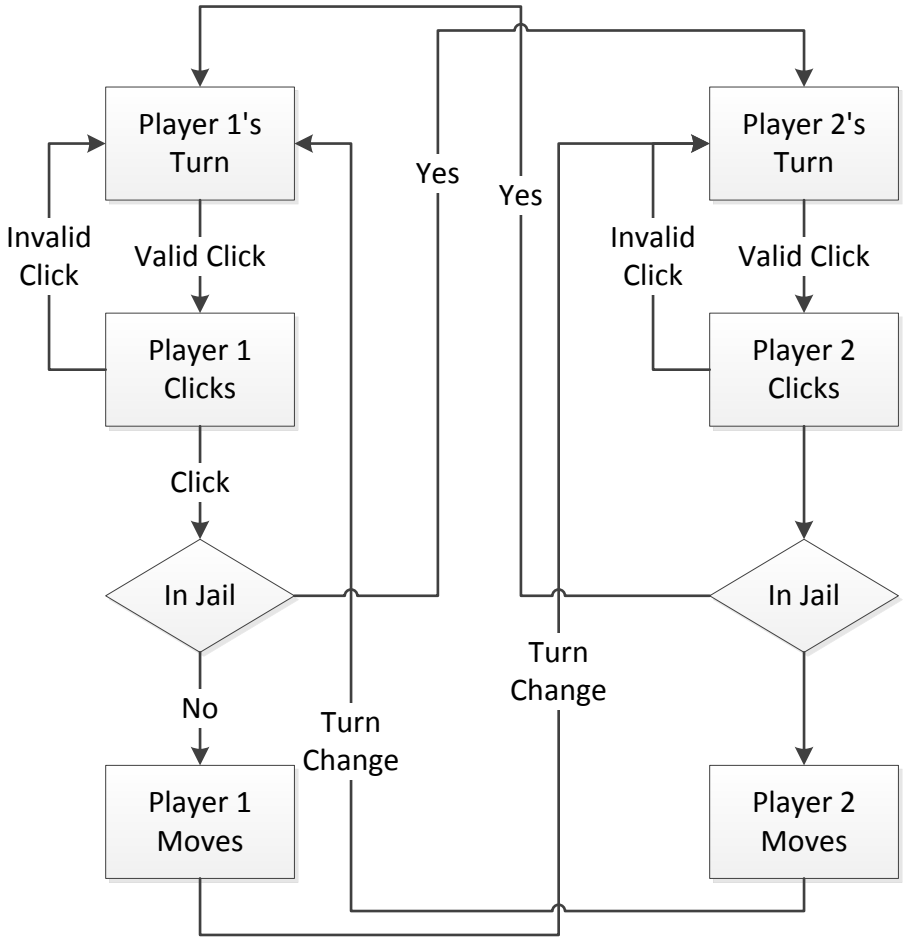
The Virtual Monopoly program is capable of supporting two-player interaction in a simplified version of the Classic Monopoly board game on the same device. Player versus computer functionality was not implemented, nor was participation by remote users on separate devices.

Domain Analysis

The original proposal included ten classes for the Monopoly program. However, six of these classes were omitted from the finished project, namely the bank, chance card, utility, community chest, railroad and tax classes, leaving the player, property, free parking and jail classes. The classes were designed around their data, with the necessary methods for retrieving and modifying their data centered on it. For instance, the Player class was identified as requiring information on the money owned by the player, the property they own and where they are on the board, as well as their name. It was decided that none of the classes naturally should inherit/extend from the others, though the free parking and jail classes may have been made to do so. However, this would have most likely just added more data and methods to the property class unused by most property objects. Thus, all classes are largely independent of each other. See below for the UML diagrams of the classes.

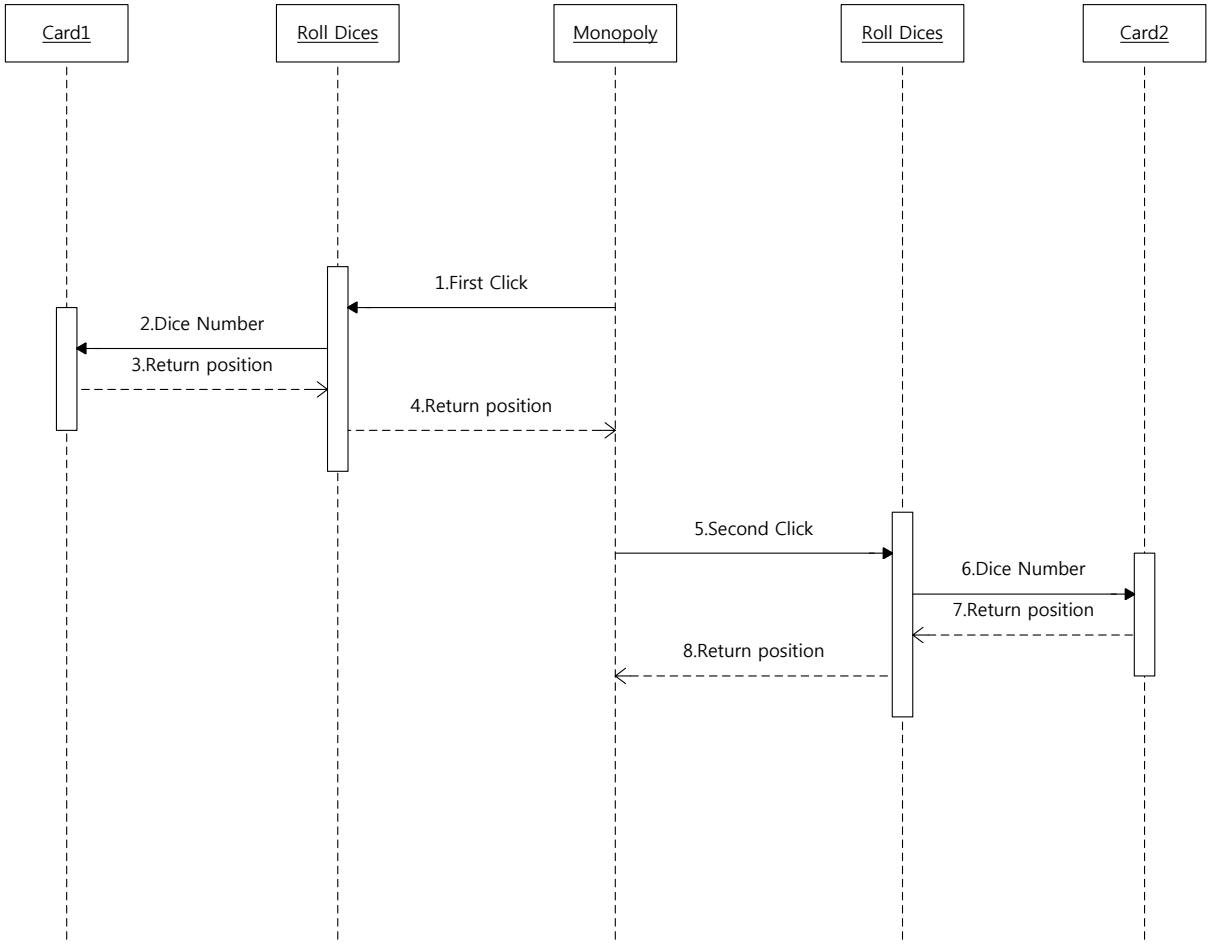

Player
-name: String
-jail: boolean
-location: int
-money: int
-doubleCount: int
-isBankrupted: boolean
+Player()
+setName(str: String) void
+getName(): String
+setProperty(prop: Property): void
+addProperty(prop: Property): void
+getProperty(): ArrayList<Property>
+setJail(jail: boolean): void
+getJail(): boolean
+setLocation(i: int): void
+getLocation(): int
+setMoney(i: int): void
+addMoney(i: int): void
+takeMoney(i: int): void
+getMoney(): int
Property
-name: String
-color: String
-price: int
-owned: boolean
+Property()
+setName(str: String): void
+getName(): String
+setColor(str: String): void
+getColor(): String
+setPrice(i: int): void
+getPrice(): int
+setOwned(own: boolean): void
+getOwned(): boolean
MonopolyInterface
-card: JPanel[]
-parkCard: JPanel
-jailCard: JPanel
-goCard: JPanel
-goJailCard: JPanel
-player1: JPanel
-player2: JPanel
-prop: Property[]
-jail: Jail
-FP: FreeParking
-turn: int
-x: int
-y: int
-Double: boolean
-p1: Player
-p2: Player
-lb1: JLabel
-lb2: JLabel
-infoPanel: JPanel
-infoPanel2: JPanel
-info: JLabel
-info2: JLabel
-money1: JLabel
-money2: JLabel
-jailcount1: int
-jailcount2: int
+MonopolyInterface(windowTitle: String, p1: Player, p2: Player, pr: Property[], jail: Jail, FP: FreeParking)
+buildButton(): void
+buildEmpty():void
+buildSpecial(name: String): JPanel
+buildInfo(): void
+buildInfo2(): void
+buildCard(c: Color, price: int, name1: String, name2: String, position: int): JPanel
+buildPlayer1(c: Color, price: int, name1: String,
Jail
-turns: int
-payment: int
-payed: boolean
+Jail()
+TurnOver(): void
+getTurns(): int
+Paying(i: int): void
+setPayment(i: int):void
+getPayment(): int
+IsPayed(): boolean
FreeParking
-money: int
+FreeParking()
+setMoney(i: int): void
+getMoney(): void
Player buys Property
0.. 1
0.. 1
1
1
Rolls Dice
Game board consists of Property
1
1
1
Update Color to match Player
1
1
Number of turns to stay in jail
1
0.. 1
Next Player's turn if in jail, or not doubles
1
Player collects from Free Parking
1
1 Puts Player in Jail
1
Reset Free Parking money
1
1

State Model



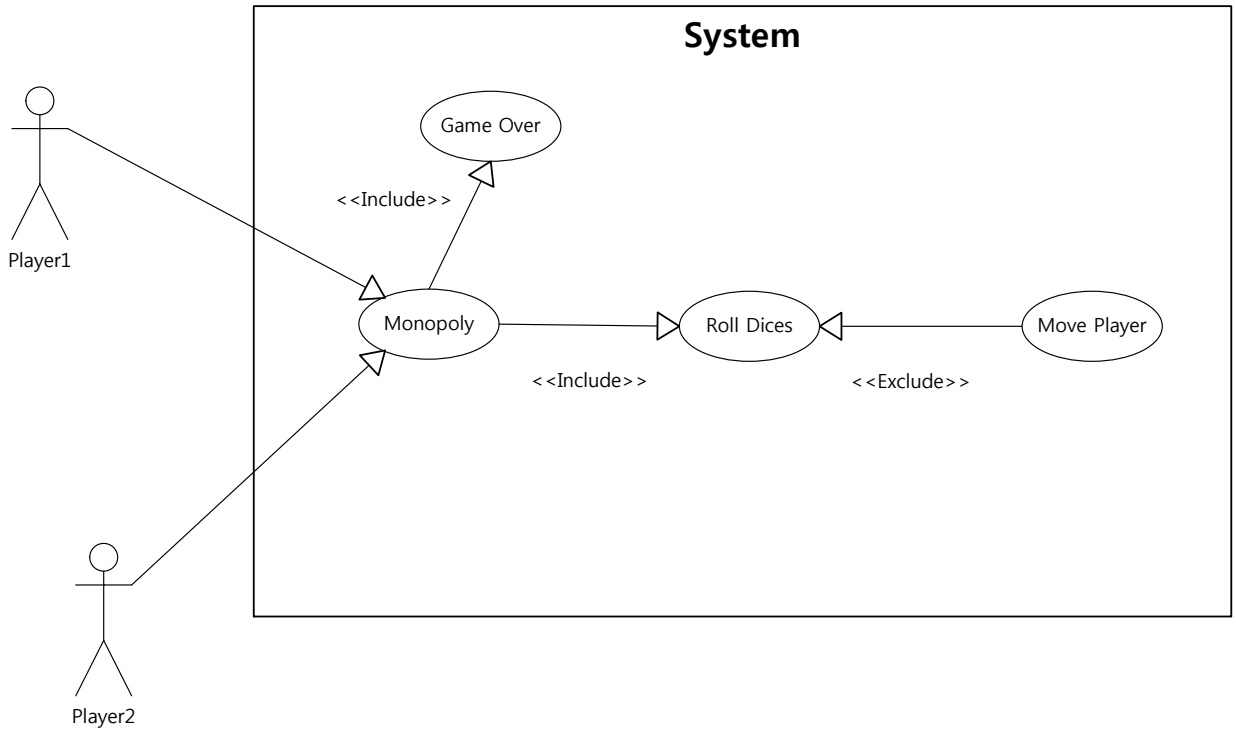
Inside the MonopolyInterface class, we have an instance that distinguished the turn of the player. It's been declared as integer type in order to leave room for future implementation. When it's player 1's turn, player 1 has to press the 'roll dice' button to roll the dice. If player 1 does not click, program waits until they click. Once player 1 clicks the button, the program rolls two dice to decide how far the player has to move. However, after rolling the dice, if the player is found to be staying in the jail, the player cannot move. In this case, the player's turn is over and turn is passed to player 2. If the player isn't in jail, they move and buy the property that they land on, pay rent if they land on the other player's property, keep their money if landing on their own property, or collect money for passing go or landing on free parking.

Sequence Diagram



The sequence diagram shows how the program is actually executing the inputs when the user inputs the clicks. First click will roll the dice and output the dice value which will bring the player to a certain card position, and from there the position is returned to the system and the player is now placed on the card position. Same logic goes with the second click.

Use Case



For now, as the program exists, there will only be two players who can play the game at a time. Also, neither player can be computer-controlled so this game is meant to go with two persons at a time to play. The Monopoly case includes two cases; Game Over and Roll Dices. The ‘Game Over’ case is the case where one of the players is bankrupted and game is over and player should reopen the game to reply. The ‘Roll Dices’ case is the case where one of the players is rolling the dice to move on the board.

Implementation

Composition

The monopoly board consists of 26 card objects. The cards are different classes; Property, Jail, and Free Parking. All these classes exist to compose one class monopoly. They have a strongly aggregated relation in a sense.

The system itself is in a very easily modifiable condition since lots of different versions of monopoly exist in the market; we didn't want to restrict our version to just the classic version. Each card has different properties that can be modified by the programmer in the future to make different versions. There properties are color, name, and price.

The system and player have to know where the player is and to where the player is going. To do this, each card is assigned a different card number starting from index 0. Once the dice output the result, a huge case loop executes to figure out what to do. For example, if the dice result is 7, system looks up at case 7 to figure out what to do and where to position the player.

GUI Implementation

The GUI for the project was implemented by creating MonopolyInterface class that extends JFrame and implements Action Listener. The basic layout of the board is written in the constructor of Monopoly Interface class, where size, name, layout type (FlowLayout) and cards were specified. Each card is constructed using the method buildCard that returns JPanel object. Cards use Border layout. Properties have different colors, which we implemented as JLabel objects with background color set to corresponding property color. The same technique was used to mark properties owned by some player: the background of the card was set to the color of the owner. All other objects on the board were implemented by the designated methods that returned the JPanel. To make an empty space in the center of the board we used empty JPanels.

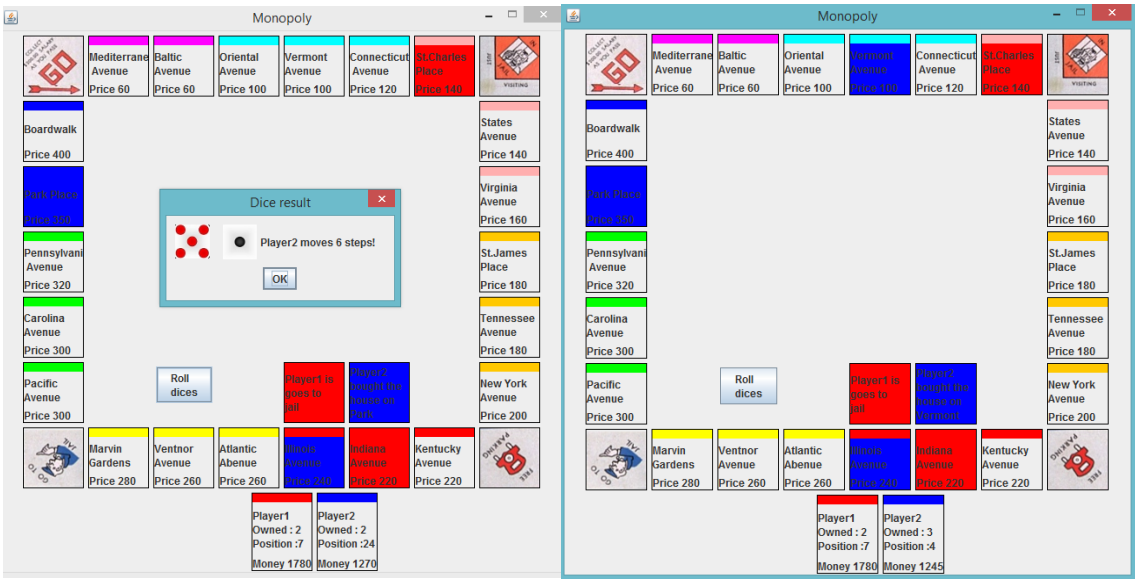
The coding regarding the game process was done in the actionPerformed method. Rolling the dices was implemented using the Random class from java.util.Random package. To generate numbers from 1 to 6, we called the method nextInt() and added 1 to its result, since it returns numbers starting from zero. We output the pictures of two dices in the messageDialog, by adding

one picture to the text of the message, and the second picture as an icon of the message. Change of turns between players was supported by having an integer variable turn. Zero value of the turn corresponded to first player, value of one to the second player. In case of double the turn didn't switch from 0 to 1 and vice versa, but stayed at the same value. The jail was also implemented by integer variable jailcount, that was set to 3 when player landed on "Go to jail" card. This count is decremented each turn, when it is zero player can roll dices again. Special cards were handled by if else statements, and corresponding messages were outputted to the screen by JOptionPane.showMessageDialog() method. To make the game process clearer and easier we added 2 information labels for each player. 1 of them contains amount of money, number of owned properties and position. The second one displays the information about the last move of a player: position and action performed. The end of the game is determined by calling the method of player class: isBankrupted().

Testing and results

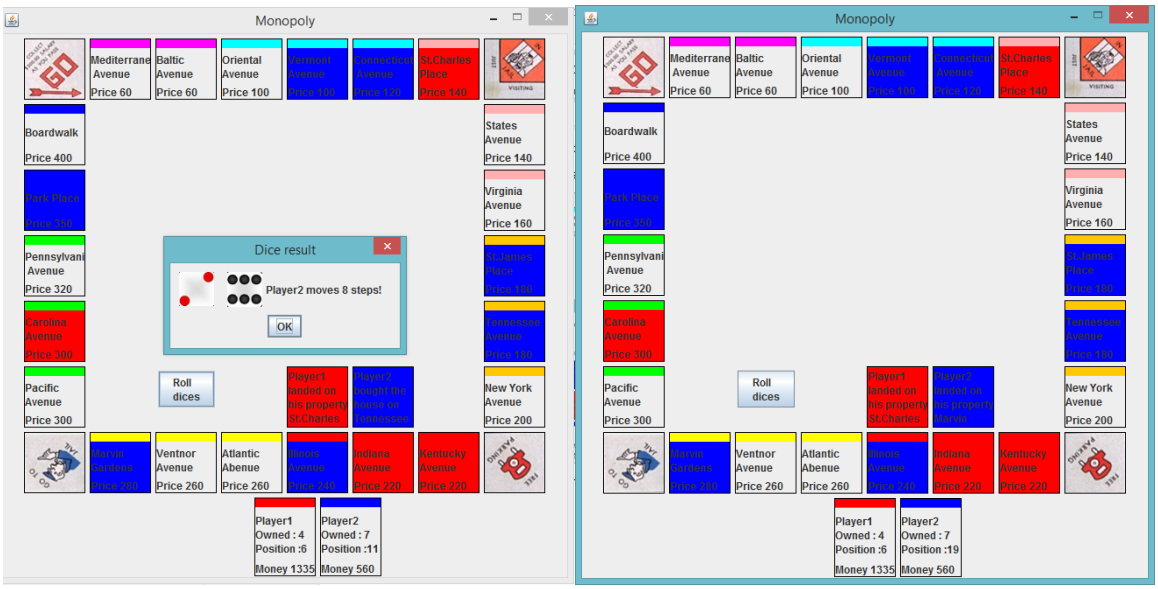
We tested our project to see if it is functioning and encountered no errors. The tests were successful.

The first test was to check that player moves the right number of steps according to the faces of the dice generated by a random number generator. Player class has attribute position, which varies from 0 to 25. Below is the check to see if position is updated correctly when player passes zero position.

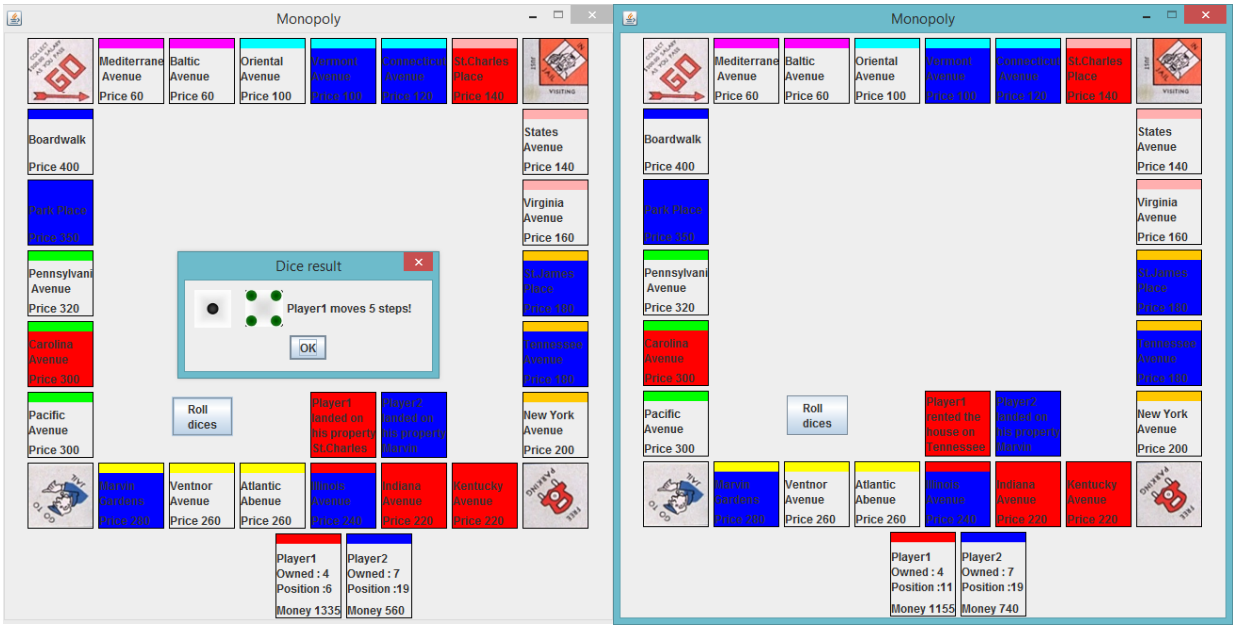


Player2 was at Park Place and moved to Vermont Avenue. This is correct change, Player2 moved 6 steps.

The next test checks the case when a player lands on his own property. In this case no money should be subtracted from the player and a corresponding message is displayed. The required behavior is shown below.



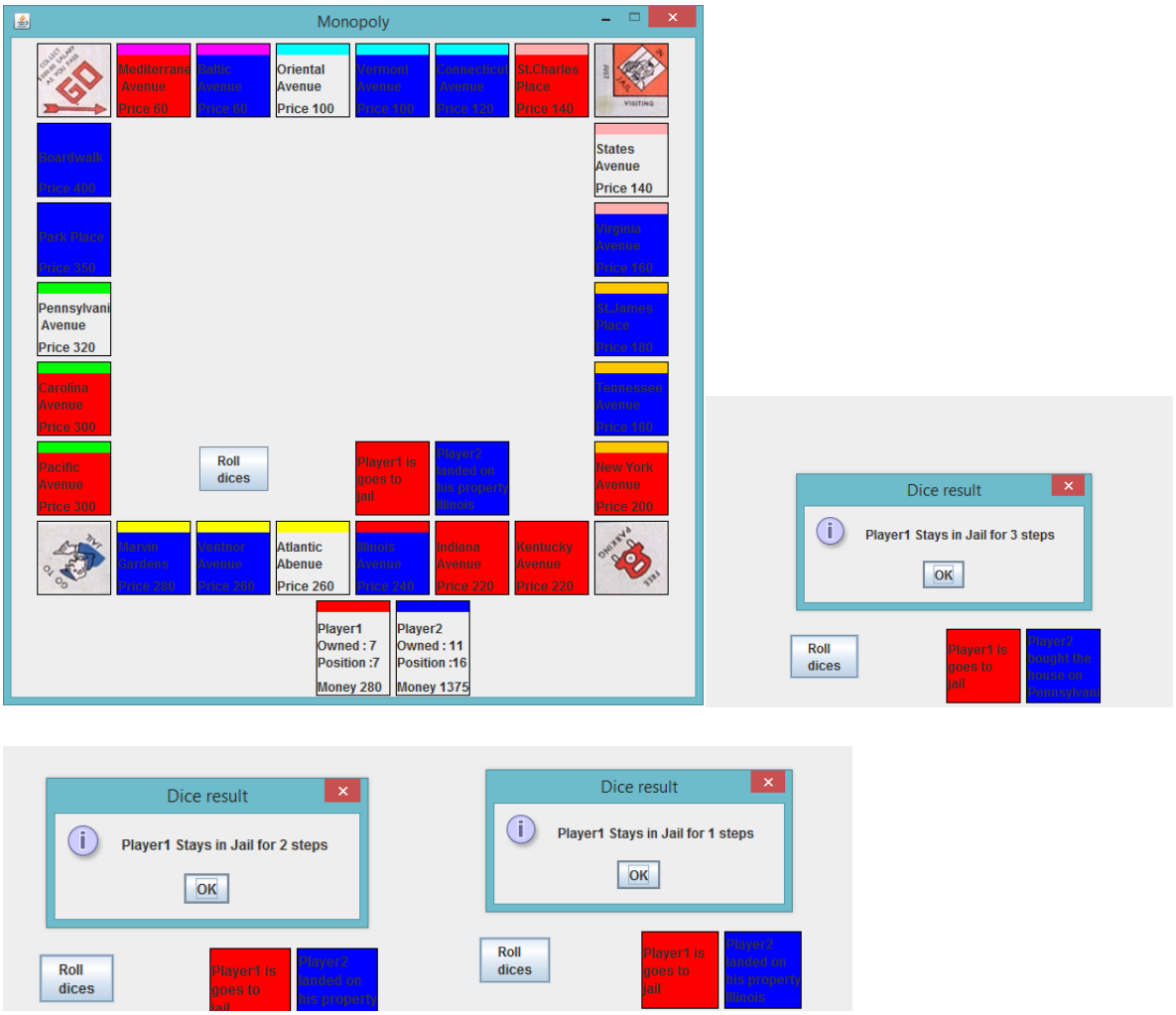
This test shows the successful renting of a place. If player lands on the property of another player, the former one should pay rent corresponding to the price of the property. On the screenshots below it can be seen that that amount of money for Player 1 has decreased and Player 2's increased. The red box says "Player1 rented the house on Tennesse".



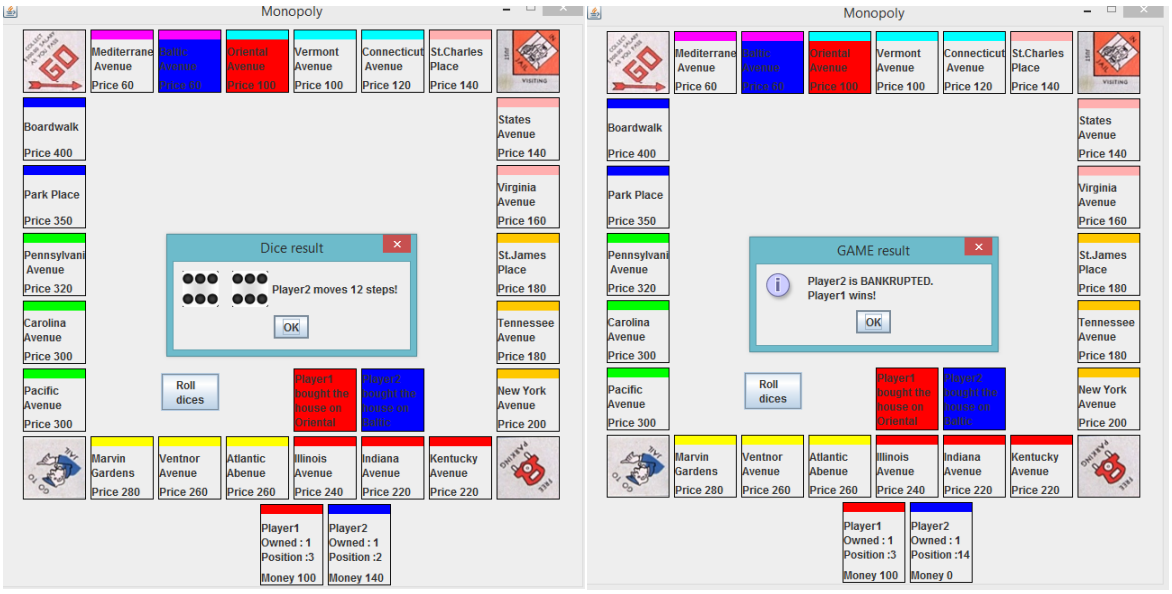
If two dice show the same face when user rolls the dice, the next turn to go will be his again.

In this case player1 had doubles twice in row. As a result player 1 made 3 moves until the result of rolling dice was not double. In addition, from the first and second screenshots we can see that amount of player1's money increased by 200 when he landed on Free Parking. This means that free parking case works successfully.

The test for Jail was successful also. When player lands on "Go to Jail" card, he skips additional 3 turns. Below the corresponding case is displayed.



The last test is for the end of the game. When one of the users goes bankrupt, the game ends.



Here, Player 2's amount of money was 140. Player 2 moves 12 steps, where he has to buy Kentucky Avenue house, which costs 220. Player 2 is not able to buy it, so he goes bankrupt, and player1 wins the game. The JOptionPane message dialog is displayed. The program exists even after a player went to bankruptcy therefore the player has to exit entire program to restart the game.

Extensibility

Monopoly game is difficult to implement because it has a large set of rules. Our team was limited on time, so many of the aspects of the game were left unimplemented. The most important thing to extend in our project is to increase the functionality. Our design doesn't cover Monopoly features such as: types of properties (railroads, utilities), chance and tax cards, giving players a choice to buy or to ignore vacant property, a bank, consequently no mortgages and loans. These features can be programmed by adding a new classes for bank, utilities, railroads and introducing appropriate methods to support the functionality of those classes; extending the functionality of the ActionPerformed method in the GUI class. The next desired extension is related to the object-oriented design. We don't use inheritance, overriding, interface classes extensively. Some of these concepts are used as a part of programming the GUI. Extending our project by adding RailRoad or utility classes, would require an inheritance. Railroad and Utility classes would be children of Property class. The Property class could have such attributes as name, location and price, which are common for all subclasses. However, as you cannot build on a railroad or utility object, the methods for the House and Railroad classes would be different. Another possible extension is to add an interface to facilitate interaction between Player and Property classes. For example, this interface can contain functions for simulating the buying and renting of properties by users. Possible extensions concerning the interface include adding a user specified names and number of players. This can be done by outputting the Input Dialog in the beginning of the game, asking for the number of players and their corresponding names. It may be uncomfortable for multiple players to play on one computer. This game can be adapted to web based platforms to allow playing over the network.