1st Assignment

Salah Uddin AXC4HX
axc4hx@inf.elte.hu
Group 9

Task

4. Simulate a simplified Capitaly game. There are some players with different strategies, and a cyclical board with several fields. Players can move around the board, by moving forward with the amount they rolled with a dice. A field can be a property, service, or lucky field. A property can be bought for 1000, and stepping on it the next time the player can build a house on it for 4000. If a player steps on a property field which is owned by somebody else, the player should pay to the owner 500, if there is no house on the field, or 2000, if there is a house on it. Stepping on a service field, the player should pay to the bank (the amount of money is a parameter of the field). Stepping on a lucky field, the player gets some money (the amount is defined as a parameter of the field). There are three different kind of strategies exist. Initially, every player has 10000. Greedy player: If he steps on an unowned property, or his own property without a house, he starts buying it, if he has enough money for it. Careful player: he buys in a round only for at most half the amount of his money. Tactical player: he skips each second chance when he could buy. If a player has to pay, but he runs out of money because of this, he loses. In this case, his properties are lost, and become free to buy. Read the parameters of the game from a text file. This file defines the number of fields, and then defines them. We know about all fields: the type. If a field is a service or lucky field, the cost of it is also defined. After the these parameters, the file tells the number of the players, and then enumerates the players with their names and strategies. In order to prepare the program for testing, make it possible to the program to read the roll dices from the file..
Print out which player won the game, and how rich he is (balance, owned properties).

Analysis

Depending on the random movement, number of rounds and rolls for each player the final output can change and players can have different number of properties with different combinations.

No. of Players	Rounds	Rolls	Random Moves

Menu

The Menu class acts as the main game engine, managing the board, players, and game flow. It initializes the players, sets up their starting positions, and defines the number of rounds.
Attributes:

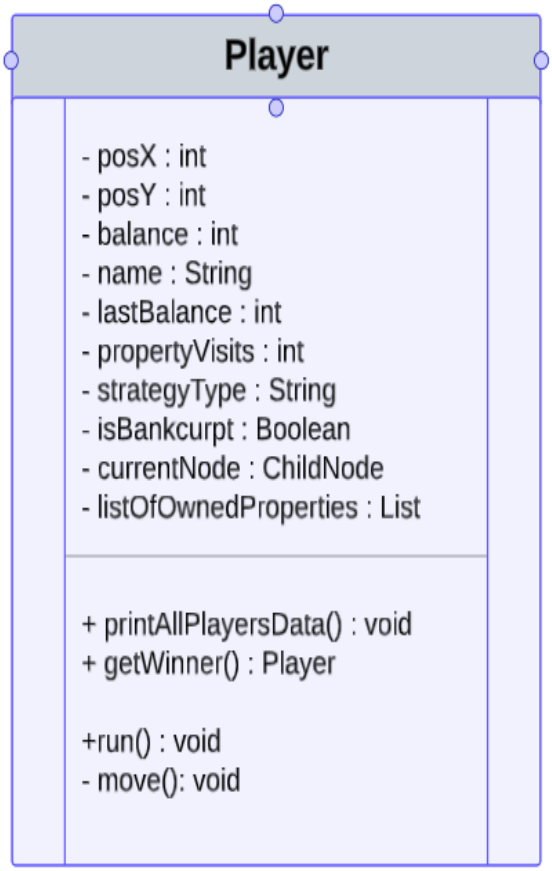
- xDim, yDim: Dimensions of the board.
- players: A list of Player objects.
- rolls: Predefined dice rolls to simulate player movement.

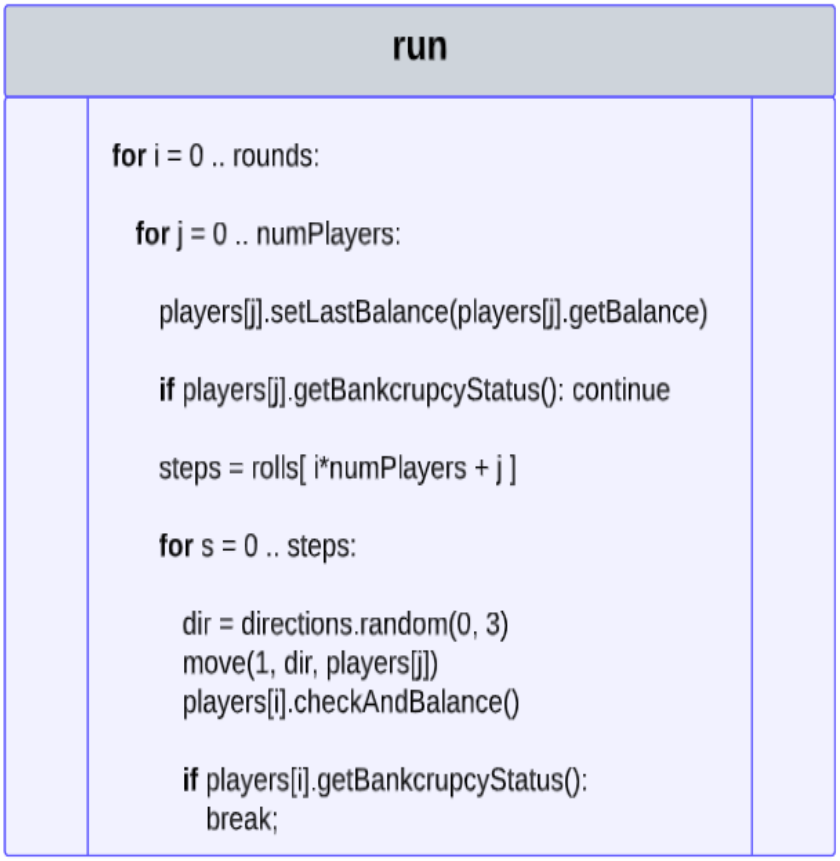
- rounds: The number of rounds in the game.
- rand: Random number generator with a seed for repeatable results.
- directions: The four possible movement directions (Up, Down, Left, Right).

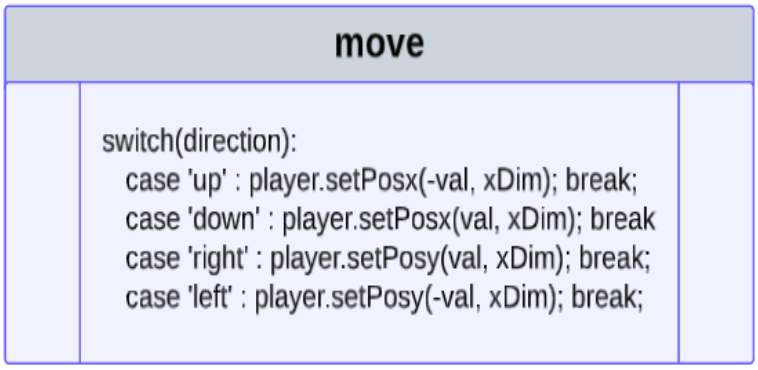
Key Methods:

- move(int val, String direction, Player player): Moves a player in the specified direction across the board.
- run(): The main method that simulates the game, executing rounds where each player moves according to dice rolls and interacts with the fields they land on.
- getWinner(): Determines and returns the player with the most properties at the end of the game.
- printAllPlayersData(): Outputs details about each player's balance, properties, and strategy type.

UML Diagrams:







CyclicDoublyLink
This class represents the doubly linked cyclic structure of the board. Each ParentNode contains a ChildNode, forming a 2D grid where players move in different directions.

Node
This is a base class for the doubly linked list representing the board. It contains references to the left and right nodes for navigation.

ParentNode
This class extends Node and contains a ChildNode list, representing the layout of fields on the board.

ChildNode
This class extends Node and from this childnode the childlist in parent, it also representing the layout of fields on the board.

RandomNumber
Utility class used to generate random numbers for player movements and dice rolls. It allows for seeded randomness for reproducibility in testing.

Player
This class defines the attributes and behaviors of each player in the game, including their movement, strategy, balance, property ownership, and interactions with fields.

Attributes:

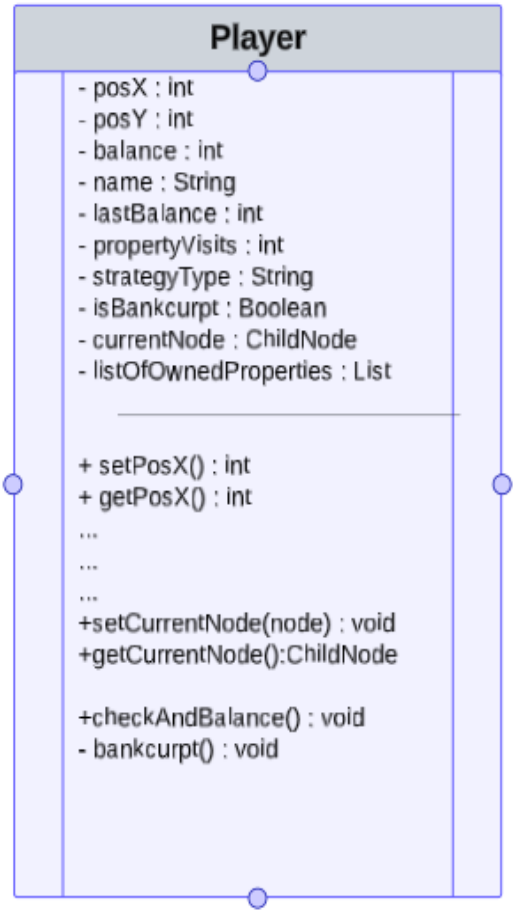
- posX, posY: The player's current position on the board.
- balance: The player's current monetary balance.
- lastBalance: Stores the player’s balance before making a transaction (useful for the Careful strategy).
- strategyType: The player's decision-making strategy (Greedy, Careful, Tactical).
- ownedProperties: A list of properties owned by the player.
- currentNode: The field (node) the player is currently located on.

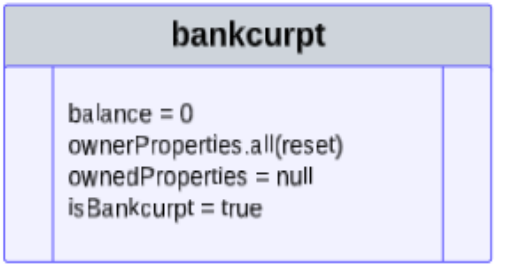
Key Methods:

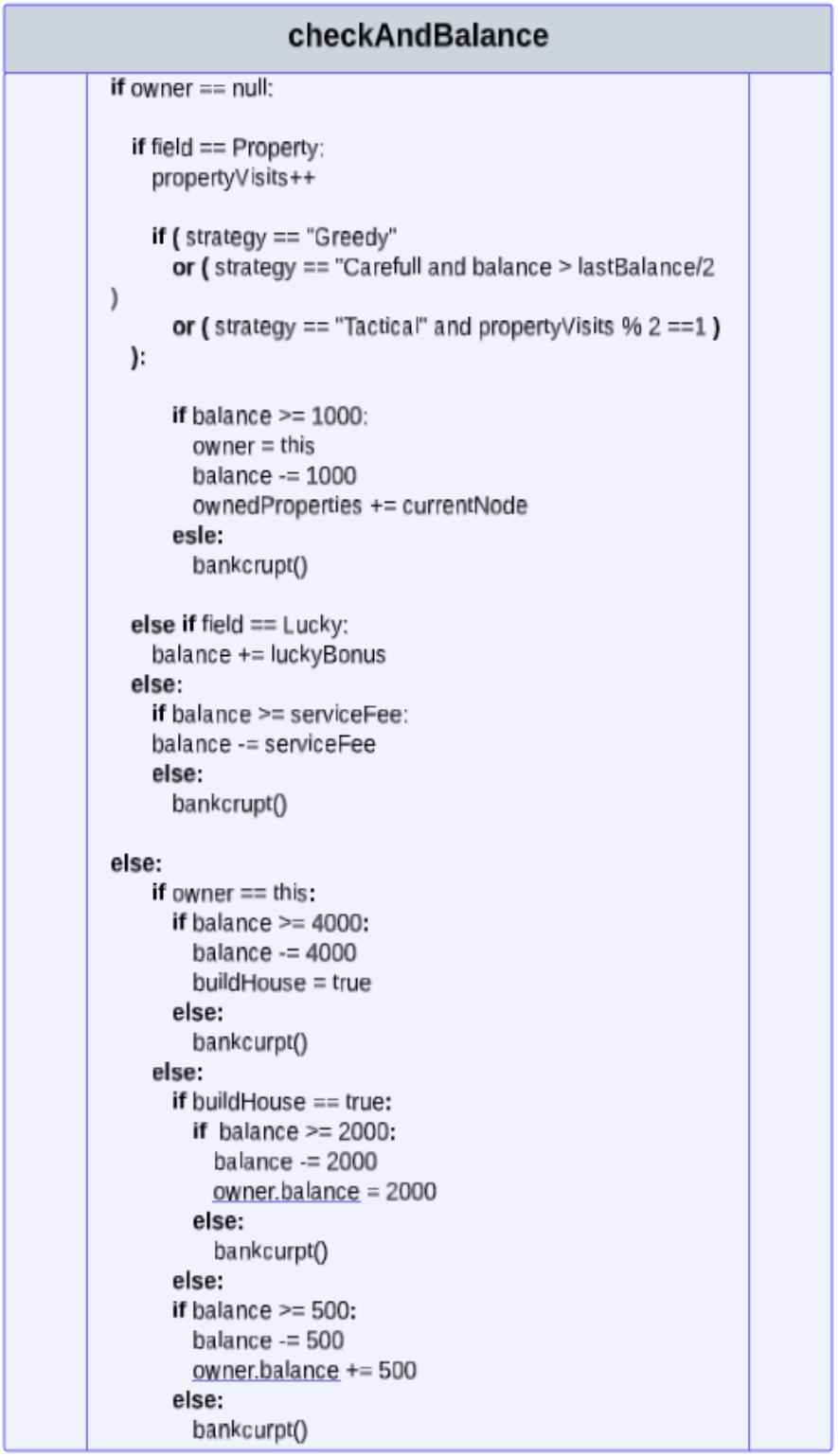
- checkAndBalance(): This method checks the field type the player lands on and updates their balance, buying properties, paying fees, or building houses accordingly.
- bankcrupt(): Handles player bankruptcy, resetting the player’s balance to 0 and freeing up their owned properties.
- addProperty(ChildNode node): Adds a property to the player's owned properties list.

- setPosX(int val, int xDim) and setPosY(int val, int yDim): Updates the player's position while accounting for the cyclical nature of the board.

UML Diagrams:







Field

This enum class defines the different types of fields on the board and their properties.

Field Types:

- Property: A field that can be bought by a player, and later a house can be built on it.
- Service: A field where players pay a service fee (e.g., 300).
- Lucky: A field that gives a bonus to the player (e.g., 500).

Field-Specific Values:

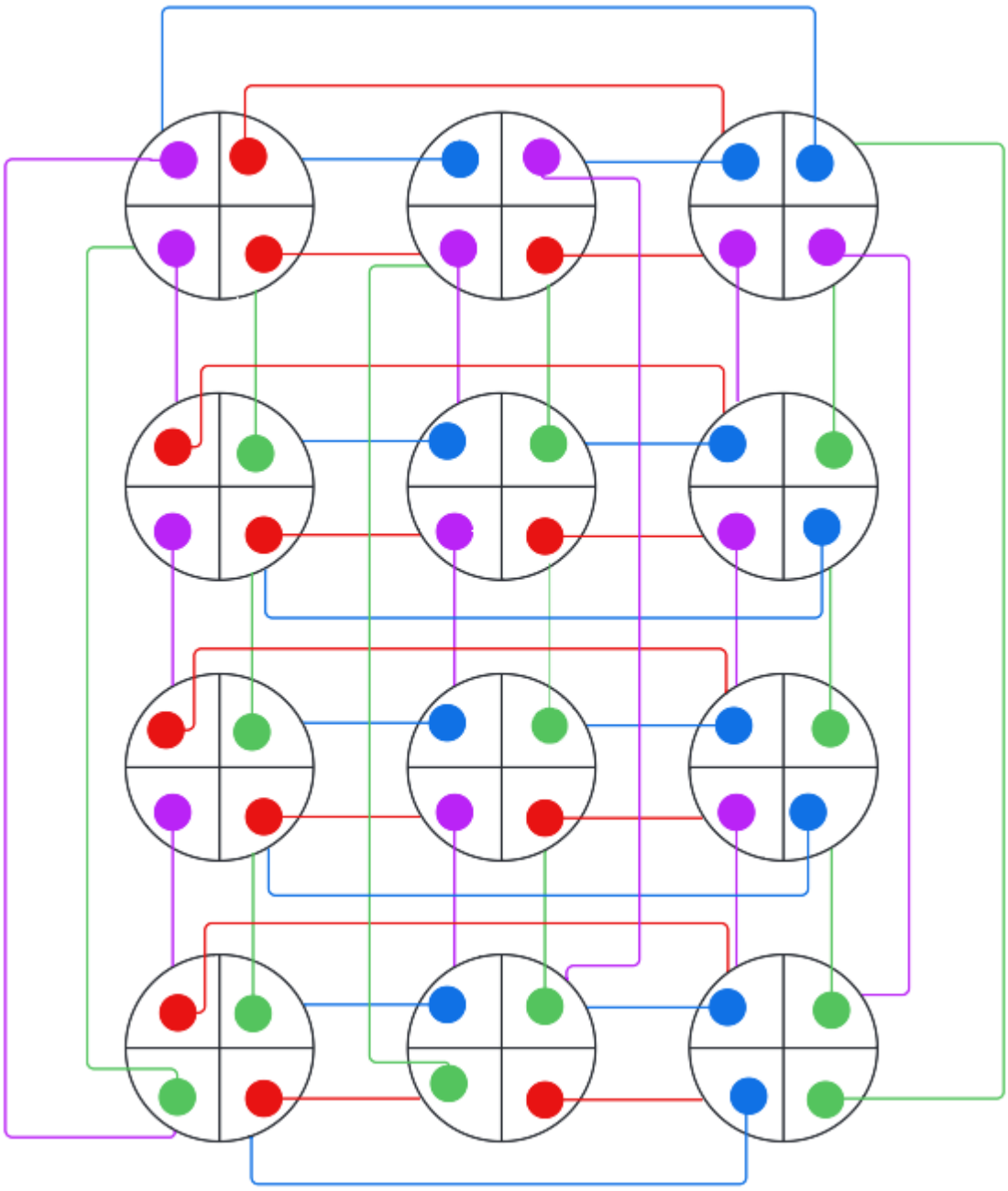
- Lucky.getLuckyBonus(): Returns the amount a player gets for landing on a Lucky field.
- Service.getServiceFee(): Returns the fee to be paid by a player landing on a Service field.

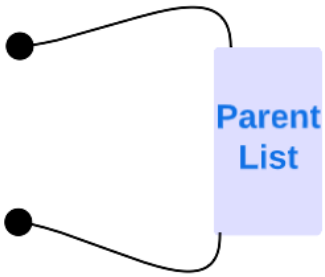
Game Flow

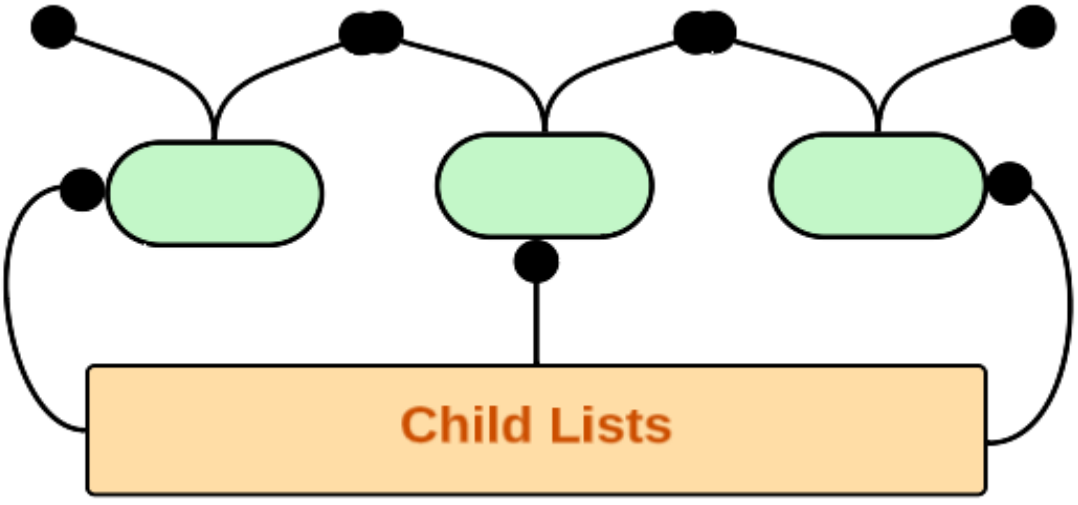
1. **Initialization**:
   The game starts by reading board dimensions and initializing the board fields based on the configuration from data.txt. Each player starts with a balance of 10,000.

2. **Player Movement**:
   Players move across the board based on dice rolls. The movement direction (up, down, left, right) is determined randomly.
3. **Field Interaction**:
   Players interact with the field they land on:
   - **Property**: The player can buy it (if unowned), build a house (if they own it), or pay rent (if another player owns it).
   - **Service**: The player pays a fee.
   - **Lucky**: The player receives a bonus.
4. **Player Strategies**:
   - **Greedy**: Buys a property if possible and builds houses aggressively.
   - **Careful**: Buys properties only if the cost is less than half of their current balance.
   - **Tactical**: Skips every second chance to buy properties.
5. **Winning Condition**:
   After all rounds, the player with the most properties wins. If multiple players own the same number of properties, the one with the highest balance wins.

Diagram:




Parent
List


Child Lists

Testing:

PlayerTest.java (JUnit Tests)

This class provides unit tests for the core functionality of the game. It checks various aspects like balance updates, property ownership, house-building, and bankruptcy scenarios. Additionally, it verifies the game winner based on defined rounds.

Key Test Methods:

1. testInitialPositionAndStrategyType(): Verifies if a player is initialized with the correct position, strategy, name, and balance.
2. testBalanceAfterLuckyField(): Tests if a player's balance increases correctly after stepping on a Lucky field.
3. testBalanceAfterServiceFee(): Tests balance deduction when a player steps on a Service field.
4. testBankruptcy(): Ensures players go bankrupt when their balance falls below zero.
5. testPropertyOwnership(): Verifies property acquisition by players.
6. testHouseBuilding(): Tests if a player can build a house on an owned property.
7. testWinner(): Runs the game and asserts the correct winner at the end.