1st Assignment -> Task 3

Miraida Saparova 10/22/2024

Task

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 what we know about each player after a given number of rounds (balance, owned properties).

Analysis

This task simulates a simplified version of the Capitaly board game, requiring an object-oriented approach to model players, fields, and game logic. The game board contains different types of fields—properties, services, and lucky fields—each with distinct interactions. Players employ different strategies (Greedy, Careful, Tactical) to decide whether to buy properties or make other moves. The game is driven by dice rolls, with players starting with a fixed balance and moving in a cyclical manner across the board. If a player runs out of money, they are eliminated, and their properties become available for others. The task includes reading the game parameters, such as board setup and player actions, from a file, making it well-suited for automated testing and simulation of various scenarios.

1st Assignment -> Task 3

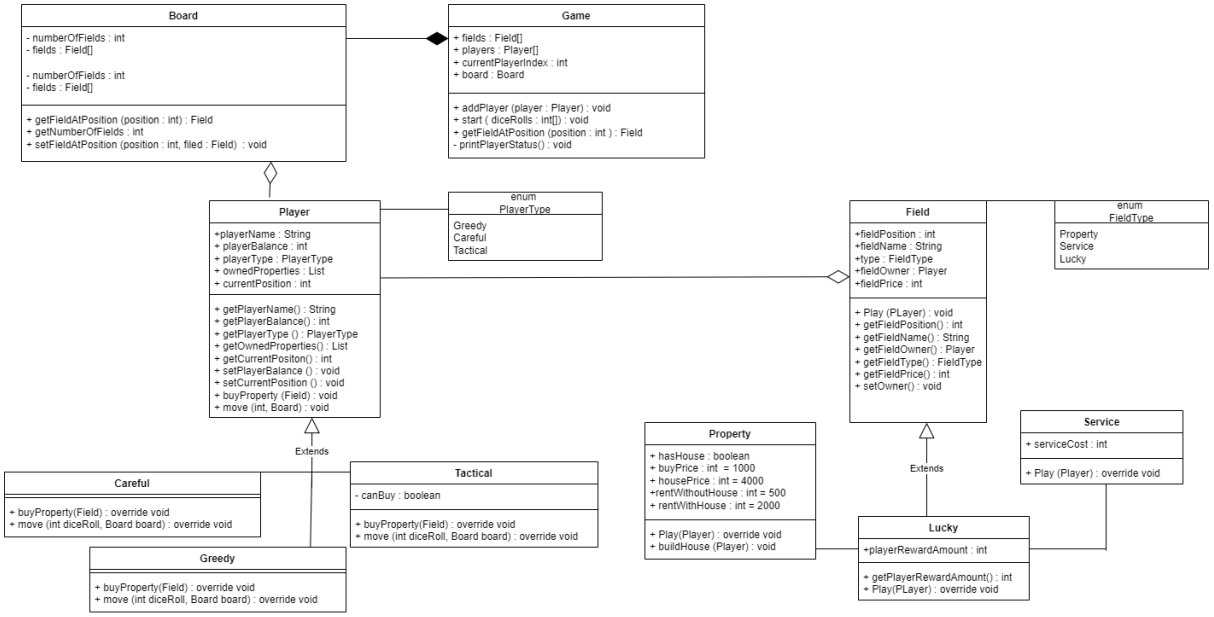
Plan

To simulate a simplified Capitaly game, several classes are introduced to represent key components of the game. The system is structured into two main parts: **fields** and **players**, each with their respective types and behaviors.

1. **Fields**:
    - **Base Class**: The `Field` class describes general properties shared by all field types, such as `id` and the ability to define specific actions when players land on them.
    - **Property Fields**: The `Property` class inherits from `Field` and represents fields that can be bought, owned, and developed with houses. It contains attributes for ownership and houses and methods for purchasing and charging rent.
    - **Service Fields**: The `Service` class represents fields where players must pay a fixed service fee to the bank. It has an additional attribute, `serviceCost`, that is charged upon landing.
    - **Lucky Fields**: The `Lucky` class provides a reward to players landing on it. It contains a `luckyReward` attribute that defines the amount players receive.
2. **Players**:
    - **Base Class**: The `Player` class models the general properties of each player, including their name, strategy, balance, and owned properties. Common methods like `move()` and `pay()` handle movement and monetary transactions.
    - **Player Strategies**: Three types of players are modeled, each inheriting from the `Player` class and implementing a different buying behavior:
        - *Greedy Player*: Always buys a property if they have enough money.
        - *Careful Player*: Buys only if the purchase doesn't exceed half their current balance.
        - *Tactical Player*: Buys only every second opportunity.
3. **Game Logic**:
    - **Board Class**: The `Board` class manages the circular nature of the game board and holds the list of fields. It ensures players move correctly and interact with fields as per the rules.
    - **Game Class**: The `Game` class orchestrates the entire game. It initializes players, manages turns, processes dice rolls, and handles bankruptcies. It contains methods to add players and track the game's progress.
4. **File Parsing**:
    - The input file is structured to first describe the fields on the board, followed by player details and a list of dice rolls.
    - The input includes:
        - Number of fields and their types (e.g., `property`, `service`, `lucky`).
        - Number of players and their respective strategies (e.g., `greedy`, `careful`, `tactical`).
        - A sequence of dice rolls that determine player movement.
5. **Object-Oriented Design**:
    - Each player strategy is implemented in its own class to respect the **Open/Closed Principle** of SOLID design. This allows easy extension if more strategies need to be introduced later.
    - Instead of conditionals for player behavior, **polymorphism** is used to let each strategy handle decisions (e.g., when to buy a property) independently.
6. **Bankruptcy Handling**:
    - If a player's balance drops below zero, they lose the game. Their properties become available for purchase, and they are removed from the game.
    - This is managed within the `Game` class, which tracks player eliminations and ensures smooth continuation of gameplay.
7. **Testing and Validation**:
    - Multiple test cases are created to simulate different board setups and dice rolls, ensuring the game logic handles various scenarios, including edge cases like multiple bankruptcies in a single round.

1st Assignment -> Task 3

UML Class Diagram



Testing

<u>Black-Box Testing</u>

1. Test Case 1: Valid Game Initialization
    - Input: A well-structured `input.txt` file with 5 fields (3 properties, 1 service, 1 lucky), 3 players (with different strategies), and a series of dice rolls.
    - Expected Output: The game initializes successfully, with all players starting with a balance of 10,000 and the fields properly assigned.
2. Test Case 2: Property Purchase by Greedy Player
    - Input: Dice rolls that lead the greedy player to land on an unowned property.
    - Expected Output: The player should purchase the property if they have enough balance (>= 1,000). Verify that the property's owner changes.
3. Test Case 3: Rent Payment on Owned Property
    - Input: One player lands on a property owned by another player that has no house.
    - Expected Output: The player pays 500 to the property owner. Verify the balances of both players are updated correctly.
4. Test Case 4: Lucky Field Interaction
    - Input: A player lands on a lucky field.
    - Expected Output: The player receives the specified lucky amount. Verify that their balance is updated accordingly.

1st Assignment -> Task 3

5. Test Case 5: Insufficient Funds to Pay Rent
    - Input: A player lands on an owned property with rent that exceeds their current balance.
    - Expected Output: The player goes bankrupt, losing their properties. Verify the properties are available for purchase.

White-Box Testing

6. Test Case 6: Dice Roll Logic
    - Input: A series of predetermined dice rolls.
    - Expected Output: Verify that the players move correctly on the board based on the rolls, and confirm that the internal state (current position) is updated.
7. Test Case 7: Strategy Execution for Tactical Player
    - Input: A tactical player lands on a property and the next turn they land on the same property.
    - Expected Output: The player should skip the first chance to buy (if it's unowned) and attempt to buy it only on the second landing. Validate the execution path taken.
8. Test Case 8: Careful Player Purchase Logic
    - Input: A careful player lands on an unowned property with exactly 10,000 balance.
    - Expected Output: The player should not purchase the property since it exceeds their limit of spending half their balance (i.e., >= 5,000). Check that they maintain their balance.
9. Test Case 9: Multiple Property Ownership
    - Input: Players purchase properties until one player owns several properties.
    - Expected Output: Verify the properties owned by the player and ensure they are correctly recorded in the player's properties list.
10. Test Case 10: Game Continuation After Bankrupt Player
    - Input: Simulate multiple rounds until a player goes bankrupt.
    - Expected Output: Ensure the game continues with remaining players, and verify that the bankrupt player's properties are available for purchase.