Programming Technology, 1st assignment documentation

Rza Jan Mohammadi YA **1st assignment/4th task** . **15 October 2024**

Neptun ID: AV2Z3A

Mail: av2z3a@inf.elte.hu

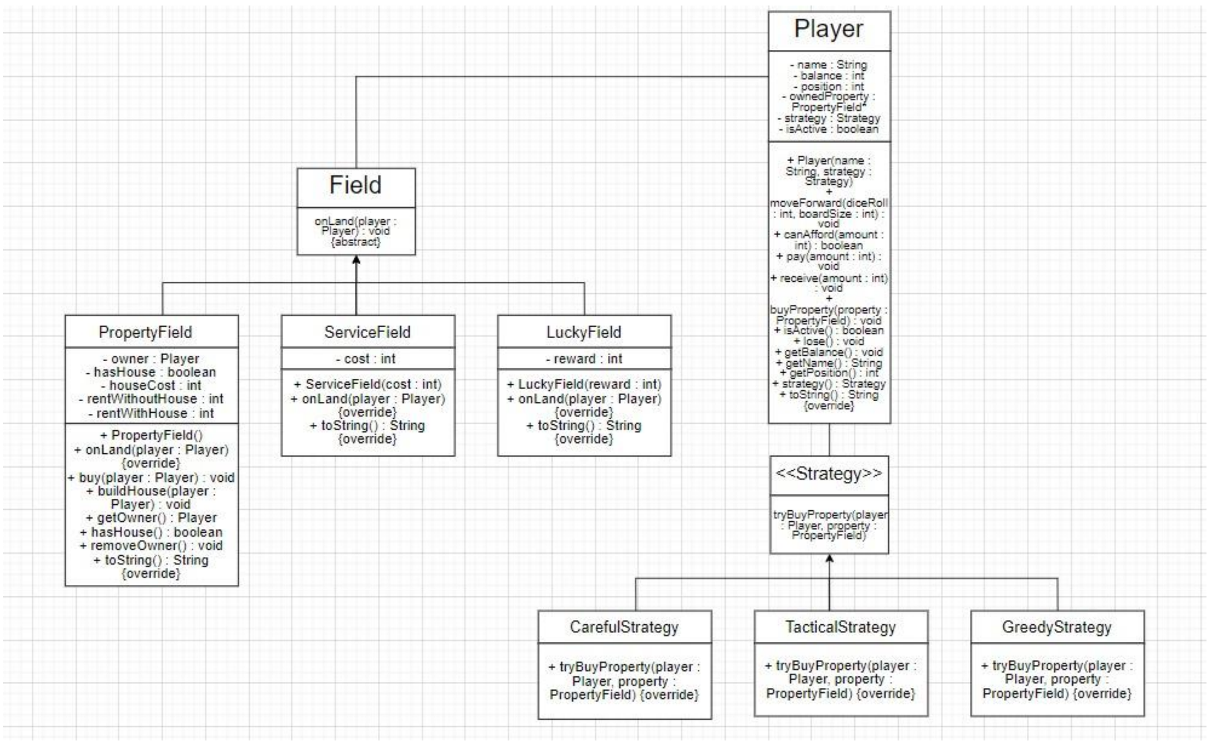
TASK

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 which player won the game, and how rich he is (balance, owned properties).

UML CLASS DIAGRAM

This diagram was made with Draw.io


Player
- name : String
- balance : int
- position : int
- ownedProperty : PropertyField*
- strategy : Strategy
- isActive : boolean
+ Player(name : String, strategy : Strategy)
+ moveForward(diceRoll : int, boardSize : int) : void
+ canAfford(amount : int) : boolean
+ pay(amount : int) : void
+ receive(amount : int) : void
+ buyProperty(property : PropertyField) : void
+ isActive() : boolean
+ lose() : void
+ getBalance() : void
+ getName() : String
+ getPosition() : int
+ strategy() : Strategy
+ toString() : String {override}
Field
onLand(player : Player) : void {abstract}
PropertyField
- owner : Player
- hasHouse : boolean
- houseCost : int
- rentWithoutHouse : int
- rentWithHouse : int
+ PropertyField()
+ onLand(player : Player) {override}
+ buy(player : Player) : void
+ buildHouse(player : Player) : void
+ getOwner() : Player
+ hasHouse() : boolean
+ removeOwner() : void
+ toString() : String {override}
ServiceField
- cost : int
+ ServiceField(cost : int)
+ onLand(player : Player) {override}
+ toString() : String {override}
LuckyField
- reward : int
+ LuckyField(reward : int)
+ onLand(player : Player) {override}
+ toString() : String {override}
<<Strategy>>
tryBuyProperty(player : Player, property : PropertyField)
CarefulStrategy
+ tryBuyProperty(player : Player, property : PropertyField) {override}
TacticalStrategy
+ tryBuyProperty(player : Player, property : PropertyField) {override}
GreedyStrategy
+ tryBuyProperty(player : Player, property : PropertyField) {override}

- Class: Field
  - onLand(player: Player):
  - This is an abstract method (likely in a base class). It's meant to be overridden by each specific field type (e.g., PropertyField, ServiceField, LuckyField) to define what happens when a player lands on this field. Since each field type behaves differently, the concrete implementation will vary.
  - toString(): String:
  - This method will return a string representation of the field. The exact format depends on the field type. For instance, it might describe the type of field, its cost/reward, or ownership status.

- Class: PropertyField (inherits from Field)
  - owner: Player:
  - The owner attribute stores the player who owns this property, or null if the property is unowned.
  - houseBuilt: boolean:
  - A boolean flag indicating whether a house has been built on the property or not. Houses affect the rent.
  - rentWithoutHouse: int:
  - This defines the rent that players must pay when they land on the property if no house is built (500 by default, as per the task description).
  - rentWithHouse: int:

    - This defines the rent that players must pay when they land on the property if a house is built (2000 by default, as per the task).
    - buy(player: Player):
    - This method allows the player to buy the property if it is unowned and the player has enough money. It will deduct the purchase amount (1000) from the player's balance and set them as the owner.
    - buildHouse(player: Player):
    - If the player already owns the property and has enough money, this method lets the player build a house, deducting the cost (4000) from their balance and setting houseBuilt to true.
    - removeOwnership():
    - This method removes the current owner of the property, setting the owner attribute to null and marking the property as unowned. It would be used if the owner goes bankrupt.
    - onLand(player: Player):
    - Overrides the Field's onLand method. When a player lands on a property, this method will determine what happens. If the property is unowned, the player can buy it. If it’s owned by someone else, the player must pay rent.
    - toString(): String:
    - Returns a string representation of the property field, likely including details such as ownership status, whether a house has been built, and the rent amounts.

- Class: ServiceField (inherits from Field)
    - cost: int:
    - Defines the cost that a player has to pay when landing on this service field. The amount will be deducted from the player's balance and paid to the bank.
    - onLand(player: Player):
    - Overrides the Field's onLand method. When a player lands on a service field, this method charges the player the specified cost and reduces their balance.
    - toString(): String:
    - Returns a string representation of the service field, likely including the cost.

- Class: LuckyField (inherits from Field)
    - reward: int:
    - Defines the reward that a player gets when they land on this lucky field. The amount is added to the player's balance.
    - onLand(player: Player):
    - Overrides the Field's onLand method. When a player lands on a lucky field, this method rewards the player with the specified amount and increases their balance.
    - toString(): String:
    - Returns a string representation of the lucky field, likely including the reward amount.

- Class: Player
    - name: String:
    - The name of the player.
    - strategy: Strategy:

- Refers to the player's strategy for buying properties (Greedy, Careful, Tactical).
- balance: int:
- The current balance of the player. Each player starts with 10,000 and can lose or gain money based on the game rules.
- properties: List<PropertyField>:
- A list that holds all the properties owned by the player.
- hasLost: boolean:
- A flag that indicates whether the player has lost (i.e., gone bankrupt).
- buyProperty(property: PropertyField):
- This method attempts to buy the given property, depending on the player's strategy. The strategy object will determine whether the player proceeds with the purchase.
- buildHouse(property: PropertyField):
- This method allows the player to build a house on a property they already own, again depending on their strategy.
- pay(amount: int):
- This method deducts the specified amount from the player's balance. If the balance becomes negative, the player loses.
- receive(amount: int):
- This method increases the player's balance by the specified amount.
- toString(): String:
- Returns a string representation of the player, likely including their name, balance, and owned properties.

- Class: Strategy (Interface)
  - tryBuyProperty(player: Player, property: PropertyField):
  - This method is an abstract definition, to be implemented by different strategies. It determines whether or not a player should buy a property, based on their strategy.

- Class: CarefulStrategy (inherits from Strategy)
  - tryBuyProperty(player: Player, property: PropertyField):
  - This method implements the logic for the "Careful" strategy. The player will only buy a property if the cost of the property is at most half of their current balance.

- Class: TacticalStrategy (inherits from Strategy)
  - tryBuyProperty(player: Player, property: PropertyField):
  - This method implements the logic for the "Tactical" strategy. The player will buy a property, but skips every second opportunity (i.e., every second time they could buy, they will choose not to).

- Class: GreedyStrategy (inherits from Strategy)
  - tryBuyProperty(player: Player, property: PropertyField):
  - This method implements the logic for the "Greedy" strategy. The player will always try to buy a property if they have enough money, regardless of the circumstances.

Tests

Testing the operations (black box testing):

1. AS A user, I WANT TO ensure that dice rolls are read correctly GIVEN a file with dice roll values WHEN the program reads the file THEN it should accurately interpret and process the dice rolls [Test the input8.txt file].
2. AS A user, I WANT TO ensure that player names and strategies are read correctly GIVEN a file with player definitions WHEN the program reads the file THEN it should accurately interpret and assign the correct names and strategies to the players [Test the input9.txt file].
3. AS A user, I WANT TO ensure that the player count is read correctly GIVEN a file with a number of players defined WHEN the program reads the file THEN it should accurately interpret the number of players [Test the input9.txt file].
4. AS A user, I WANT TO ensure that field types are read correctly and in order GIVEN a file with defined field types WHEN the program reads the file THEN it should accurately interpret the field types in the specified order [Test the input9.txt file].
5. AS A user, I WANT TO ensure that the field count is read correctly GIVEN a file with defined fields WHEN the program reads the file THEN it should accurately interpret the total number of fields [Test the input9.txt file].
6. AS A user, I WANT TO ensure that strategies work correctly for each strategy GIVEN a game simulation WHEN the program executes player strategies THEN each player's strategy should influence their actions as intended [Test the input9.txt file].
7. AS A user, I WANT TO verify that the Greedy strategy prioritizes acquiring properties over other actions GIVEN a game scenario WHEN a player employs the Greedy strategy THEN the player should consistently aim to purchase available properties [Test the input10.txt file].
8. AS A user, I WANT TO confirm that the Tactical strategy maintains a strategic approach to spending and saving GIVEN a game scenario. WHEN a player uses the Tactical strategy, THEN the player should make calculated investments, spending wisely and skip every second step. [Test the input11.txt file].

Testing based on the code (white box testing)

1. AS A user, I WANT TO load a file GIVEN that all fields in the file are lucky fields (L) WHEN the program attempts to read the file THEN it should throw an AllFieldsCannotBeLuckyFieldsException [Test the input1.txt file].
2. AS A user, I WANT TO load a file GIVEN that the field size is 0 or less WHEN the program attempts to read the file THEN it should throw an InsufficientFieldSizeException [Test the input2.txt file].
3. AS A user, I WANT TO load a file GIVEN that the number of players is less than 2 WHEN the program attempts to read the file THEN it should throw an InsufficientNumberOfPlayersException [Test the input3.txt file].

4. AS A user, I WANT TO load a file GIVEN that the number of players is not an integer WHEN the program attempts to read the file THEN it should throw an InvalidInputException [Test the input4.txt file].
5. AS A user, I WANT TO load a file GIVEN that there is no number near the lucky or service fields WHEN the program attempts to read the file THEN it should throw an InvalidInputException [Test the input5.txt file].
6. AS A user, I WANT TO load a file GIVEN that there is a number near a property field WHEN the program attempts to read the file THEN it should throw an InvalidInputException [Test the input6.txt file].
7. AS A user, I WANT TO ensure that dice rolls are within the valid range GIVEN a game simulation WHEN the input contains dice rolls THEN any value greater than 6 should throw an InvalidDiceRollException [Test the input7.txt file].