Simulation

of

Capitaly: A Financial Game

Written and Implemented

By

Mohammed Efaz (ZTFK53)

3rd Semester

Programming Technology

Eötvös Loránd University

Contents

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 kinds 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 must 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 these parameters, the file tells the number of the players, and then enumerates the players with their names and strategies. To prepare the program for testing, make it possible to the program to read the roll dices from the file.

Print out which player loses as a second loser.

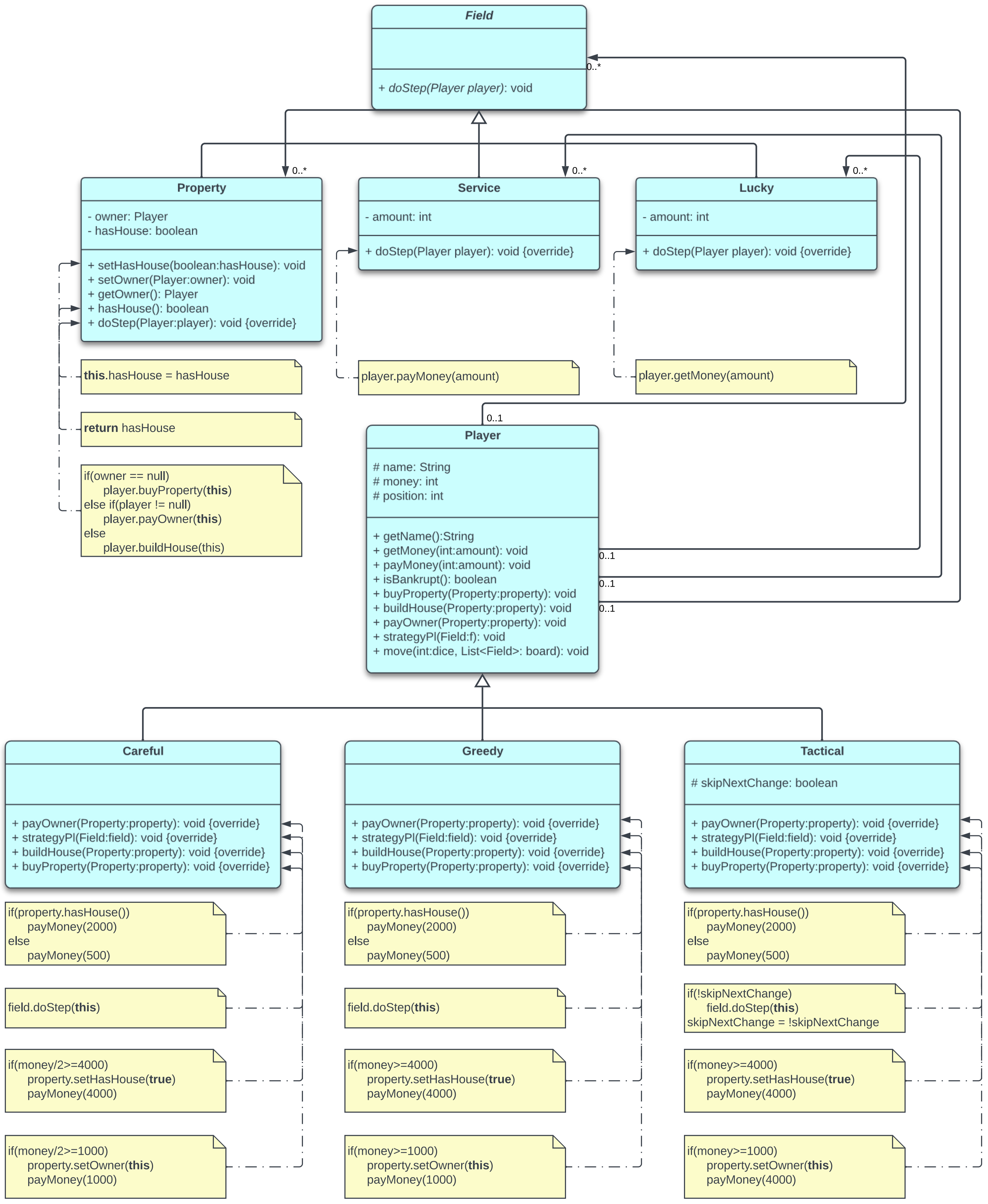
Description of the task

Two parent classes are made, `Field` (abstract) and `Player`. These will be used for inheritance later, when calling their subclasses, i.e., `Property`, `Service` and `Lucky` for the former, and `Greedy`, `Careful`, and `Tactical` for the latter. `DemoGame` is the main driver, responsible for initiating the game, simulating it, and eventually printing out the 2nd loser.

The input file is used to read both the field and player data. The dice input file gives the dice rolls for the players. All the reading is done in the `ReadFile` class. All the data (the field, players and dice numbers) are stored in their respective `ArrayLists`.

Exception handling throughout the code is used. Touched upon in details in the testing part.

UML Diagram



Field
+ doStep(Player player): void
0..*
Property
- owner: Player
- hasHouse: boolean
+ setHasHouse(boolean:hasHouse): void
+ setOwner(Player:owner): void
+ getOwner(): Player
+ hasHouse(): boolean
+ doStep(Player:player): void {override}
this.hasHouse = hasHouse
return hasHouse
if(owner == null) player.buyProperty(this) else if(player != null) player.payOwner(this) else player.buildHouse(this)
0..*
Service
- amount: int
+ doStep(Player player): void {override}
player.payMoney(amount)
0..*
Lucky
- amount: int
+ doStep(Player player): void {override}
player.getMoney(amount)
0..*
0..1
Player
# name: String
# money: int
# position: int
+ getName():String
+ getMoney(int:amount): void
+ payMoney(int:amount): void
+ isBankrupt(): boolean
+ buyProperty(Property:property): void
+ buildHouse(Property:property): void
+ payOwner(Property:property): void
+ strategyPl(Field:f): void
+ move(int:dice, List<Field>: board): void
0..1
0..1
0..1
Careful
+ payOwner(Property:property): void {override}
+ strategyPl(Field:field): void {override}
+ buildHouse(Property:property): void {override}
+ buyProperty(Property:property): void {override}
if(property.hasHouse()) payMoney(2000) else payMoney(500)
field.doStep(this)
if(money/2>=4000) property.setHasHouse(true) payMoney(4000)
if(money/2>=1000) property.setOwner(this) payMoney(1000)
Greedy
+ payOwner(Property:property): void {override}
+ strategyPl(Field:field): void {override}
+ buildHouse(Property:property): void {override}
+ buyProperty(Property:property): void {override}
if(property.hasHouse()) payMoney(2000) else payMoney(500)
field.doStep(this)
if(money>=4000) property.setHasHouse(true) payMoney(4000)
if(money>=1000) property.setOwner(this) payMoney(1000)
Tactical
# skipNextChange: boolean
+ payOwner(Property:property): void {override}
+ strategyPl(Field:field): void {override}
+ buildHouse(Property:property): void {override}
+ buyProperty(Property:property): void {override}
if(property.hasHouse()) payMoney(2000) else payMoney(500)
if(!skipNextChange) field.doStep(this) skipNextChange = !skipNextChange
if(money>=4000) property.setHasHouse(true) payMoney(4000)
if(money>=1000) property.setOwner(this) payMoney(4000)

Description of the methods

`Field (superclass)`

- `doStep(Player player)`: Abstract method. It will be inherited in the subclasses (the steps that are executed when the player lands on any of the `Property`, `Service`, `Lucky` fields).

`Property (subclass)`

- `setHasHouse(boolean hasHouse)`: Sets whether any player has any property house or not.
- `setOwner(Player owner)`: Sets the player as the owner of the property.
- `getOwner()`: Returns the owner of the property.
- `hasHouse()`: Returns true/false if the property has a house.
- `doStep(Player)`: When a player lands on property, the steps that happen; if there is no owner, the player can buy the property and can build a house. If the player is not an owner, he pays the owner (if there is an owner).

`Service (subclass)`

- `Service(int amount)`: Constructor. Sets the money.
- `doStep(Player player)`: When a player lands on service, he pays the money (to the bank).

`Lucky (subclass)`

- `Lucky(int amount)`: Constructor. Sets the money.
- `doStep(Player player)`: When a player lands on lucky, he gets the money (from the bank).

`Player (superclass)`

- `Player(String name)`: Constructor. Sets the name of the player.
- `getName()`: Returns the name of the player.
- `getMoney(int amount)`: Player's money increases by the amount. Used in the fields.
- `payMoney(int amount)`: Player's money decreases by the amount. Used in the fields.
- `isBankrupt()`: Whether the player's money is less than 0, after he pay's (or buys a house/property).
- `buyProperty(Property property)`: Implemented in subclasses.
- `buildHouse(Property property)`: Implemented in subclasses.
- `payOwner(Property property)`: Implemented in subclasses.
- `strategyPl(Field field)`: Implemented in subclasses.
- `move(int dice, List<Field> board)`: If the board is not empty, it calculates the new position by adding the dice number and then taking the remainder when dividing by the board size (for looping back to the start after reaching the end of the board). After that, the `strategyPL` method of the player's specific tactic is called.

`Greedy (subclass)`

- `Greedy(String name)`: Constructor. Sets the name of the player.
- `payOwner(Property property)`: If property has a house, player pays 2000 (to the owner), if not, 500.
- `strategyPl(Field field)`: Implements greedy tactic (using **`this`** keyword). Uses these methods.
- `buildHouse(Property property)`: If he has more than 4000, pays it for building the house.
- `buyProperty(Property property)`: If he has more than 1000, pays it for buying the property.

`Careful (subclass)`

- `Careful(String name)`: Constructor. Sets the name of the player.
- `payOwner(Property property)`: If property has a house, player pays 2000 (to the owner), if not, 500.
- `strategyPl(Field field)`: Implements careful tactic (using **this** keyword). Uses these methods.
- `buildHouse(Property property)`: If he has double the money of 4000, pays it for building house.
- `buyProperty(Property property)`: If he has double the money of 1000, pays it for buying property.

`Tactical (subclass)`

- `Tactical(String name)`: Constructor. Sets the name of the player.
- `payOwner(Property property`: If property has a house, player pays 2000 (to the owner), if not, 500.
- `strategyPl(Field field)`: Implements tactical strategy (using **this** keyword). Uses these methods. Skips every alternate chance of buying any property or building house.
- `buildHouse(Property property)`: If he has more than 4000, pays it for building the house.
- `buyProperty(Property property)`: If he has more than 1000, pays it for buying the property.

`ReadFile Class`

- `readFields(String file)`: Reads the number of fields and the field type and its amount (if any).
- `readPlayers(String file)`: Reads the number of players, their names and their tactic.
- `readDice(String file)`: Reads the numbers on the dice.

Testing

White box test cases

Type of test	Input	Expected output
`isBankrupt()`	`player.payMoney(11000)`	`Successful. Bankruptcy.`
`Careful buying Property`	`Player2.payMoney(9999)`	`Successful. Did not buy.`
`Lucky will increase money`	`new Lucky(9999)`	`Successful. Money increased.`

- Testing the `isBankrupt()` method by paying 11000.
- Testing if `Careful` player buys property if he has insufficient money.
- Testing if `Lucky` field will increase player money if he lands on said field.

Black box test cases

Type of test	Input	Expected output
Empty file content	`input3.txt`	`Empty file.`
Wrong field type	`input4.txt`	`Wrong field type.`
Wrong player type	`input5.txt`	`Wrong player type.`
Negative numbers	`input2.txt`	`Negative number is invalid.`
Default working test	`input.txt`	`Alice is the second loser.`

- Throws exception: If the .txt file is empty.
- Throws exception: If the field type is wrong, i.e., anything other than Property, Service and Lucky.
- Throws exception: If the player type is wrong, i.e., anything other than Careful, Tactical and Greedy.
- Throws exception: If the cost of the fields are negative numbers.
- The default working file: Should return an answer given the input contents are correct.