ELTE FACULTY OF INFORMATICS

DOCUMENTATION

NAME : MD FARID

NEPTUNE: WEFKHB

PROGRAMMING TECHNOLOGY

ASSIGNEMNT no .3

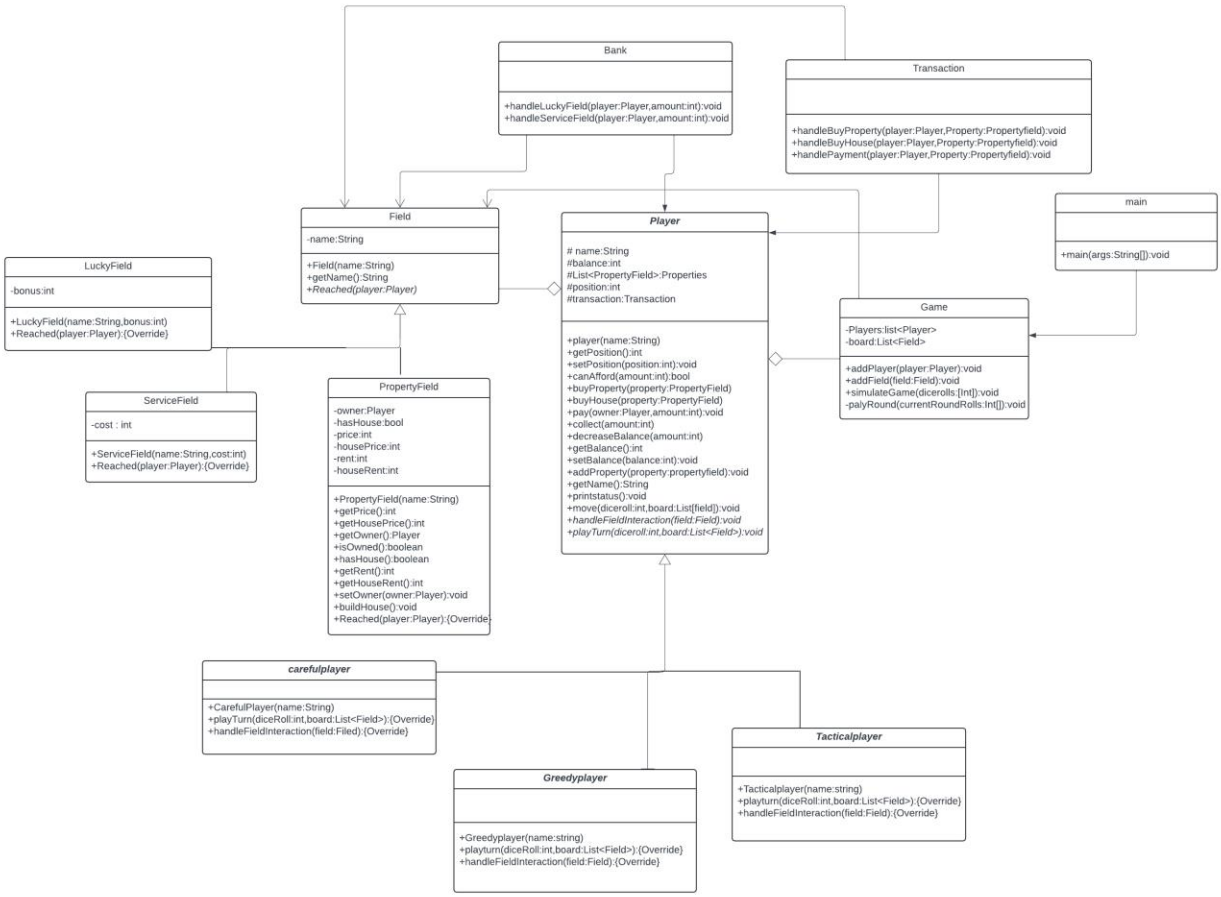
TASK

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).



inheritance
propertyfield,serviceField,LuckyField
inhertis from Field
GreedyPlayer,CarefulPlayer,TacticalPlayer
inherits from Player

Aggregation
Game aggregates Player
Game aggregates Field

Association
Game and Player
Game and Field
Transaction interacts with Player and PropertyField.
Bank interacts with Player

White Box Testing

White box testing involves examining the internal structures or workings of an application, rather than just its functionality (black box testing). It aims to ensure that all paths through the code are tested, and it's often based on the code logic itself.

1. **The check for the service fee cost being positive**: This involves internal logic to validate that service fees meet specific criteria.

Input1.txt

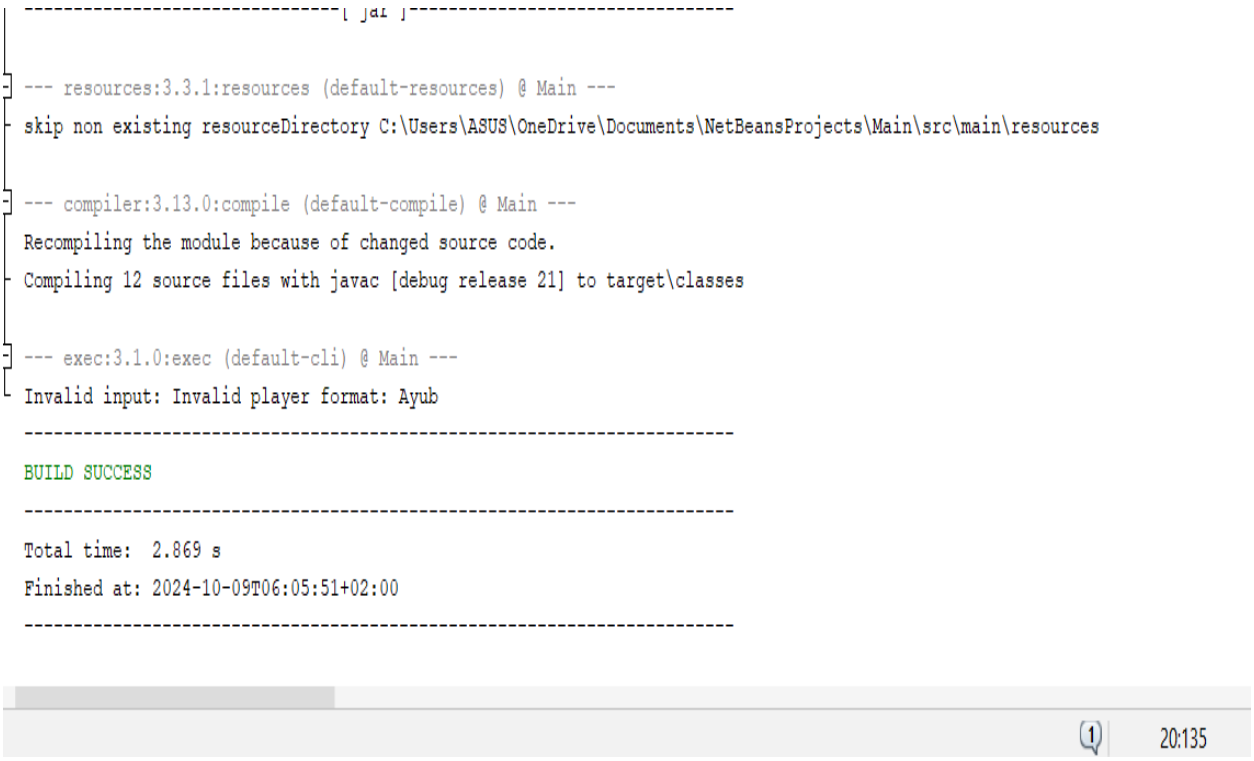
```
10
Property
Lucky 500
Service 300
Property
Lucky 1000
Service -100
Property
Property
Service 500
Lucky 200
3
Farid Greedy
Ayub Careful
Sharukh Tactical
4 2 5
6 3 4
2 5 1
```

```
--------------------------------[ jar ]---------------------------------


--- resources:3.3.1:resources (default-resources) @ Main ---
skip non existing resourceDirectory C:\Users\ASUS\OneDrive\Documents\NetBeansProjects\Main\src\main\resources

--- compiler:3.13.0:compile (default-compile) @ Main ---
Recompiling the module because of changed source code.
Compiling 12 source files with javac [debug release 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ Main ---
Invalid input: Service field cost must be positive: -100
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.414 s
Finished at: 2024-10-09T06:02:05+02:00
------------------------------------------------------------------------
```

2. **Invalid player format (first write the name and then the strategy of the player)**: This validation checks the internal structure of player data being read, ensuring it follows the expected format.

Input2.txt
15
Property
Lucky 500
Service 300
Property
Lucky 1000
Service 200
Property
Property
Service 500
Lucky 200
Service 200
Property
Property

Service 500
Lucky 200
3
Farid Greedy
Ayub
Sharukh Tactical
2 5 6
6 3 4
2 5 1
4 5 5
2 4 5



3 **Checking for unique name and unique strategy**: This involves logic to enforce game rules that prevent duplicate strategies for the same player, indicating an understanding of how player data is handled internally.

Input3.txt

10

Property

Lucky 500

Service 300

Property

Lucky 1000

Service 200

Property

Property

Service 500

Lucky 200

3

Farid Greedy

Farid Greedy

Ayub Careful

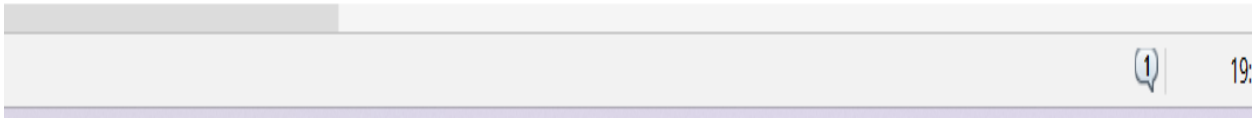
Sharukh Tactical

4 2 5

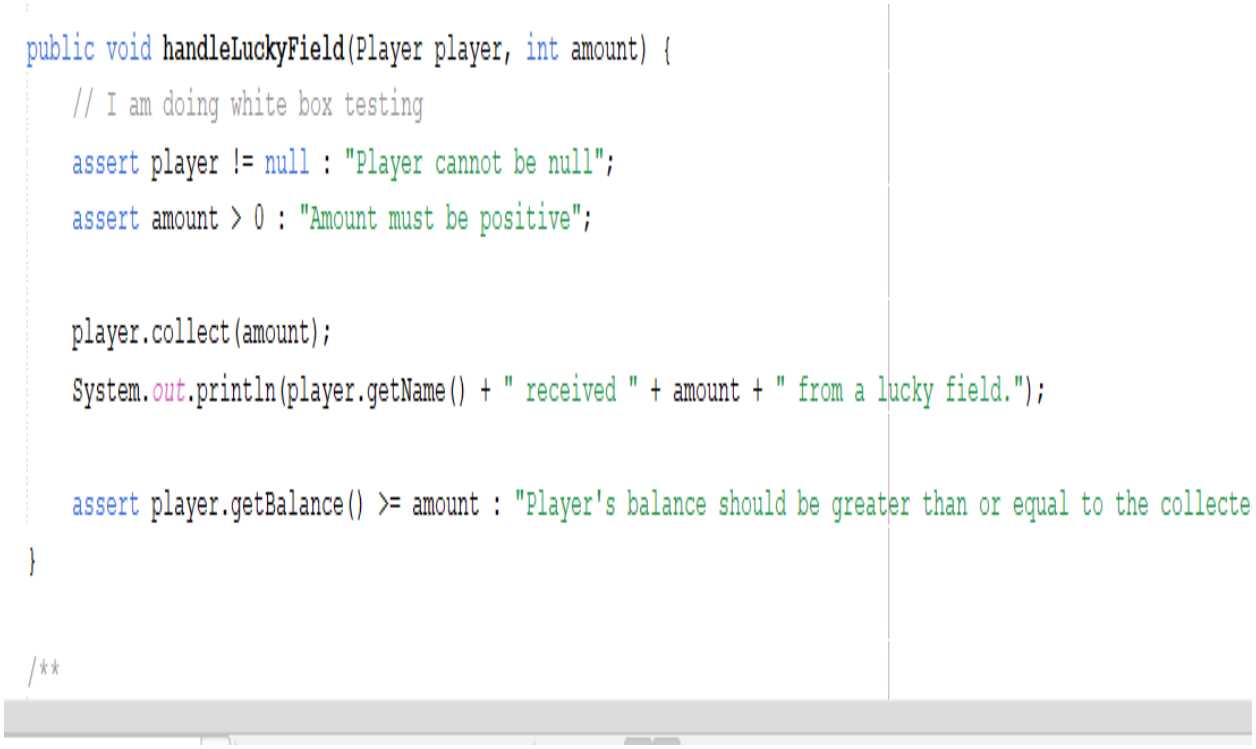
6 3 4

2 5 1

```
--- resources:3.3.1:resources (default-resources) @ Main ---
skip non existing resourceDirectory C:\Users\ASUS\OneDrive\Documents\NetBeansProjects\Main\src\main\resources

--- compiler:3.13.0:compile (default-compile) @ Main ---
Recompiling the module because of changed source code.
Compiling 12 source files with javac [debug release 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ Main ---
Invalid input: Duplicate player entry: Farid-Greedy
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.069 s
Finished at: 2024-10-09T06:08:04+02:00
------------------------------------------------------------------------
```



4.**Input Validation** : Assertions check for non-null players and positive amounts, ensuring valid inputs before executing method logic in `Bank`.



```
public void handleLuckyField(Player player, int amount) {
    // I am doing white box testing
    assert player != null : "Player cannot be null";
    assert amount > 0 : "Amount must be positive";

    player.collect(amount);
    System.out.println(player.getName() + " received " + amount + " from a lucky field.");

    assert player.getBalance() >= amount : "Player's balance should be greater than or equal to the collecte
}

/**
```

5 **dice range** : the dice range should be in between 1 and 6.

Input4.txt

10

Property

Lucky 500

Service 300

Property

Lucky 1000

Service 200

Property

Property

Service 500

Lucky 200

3

Farid Greedy

Ayub Careful

Sharukh Tactical

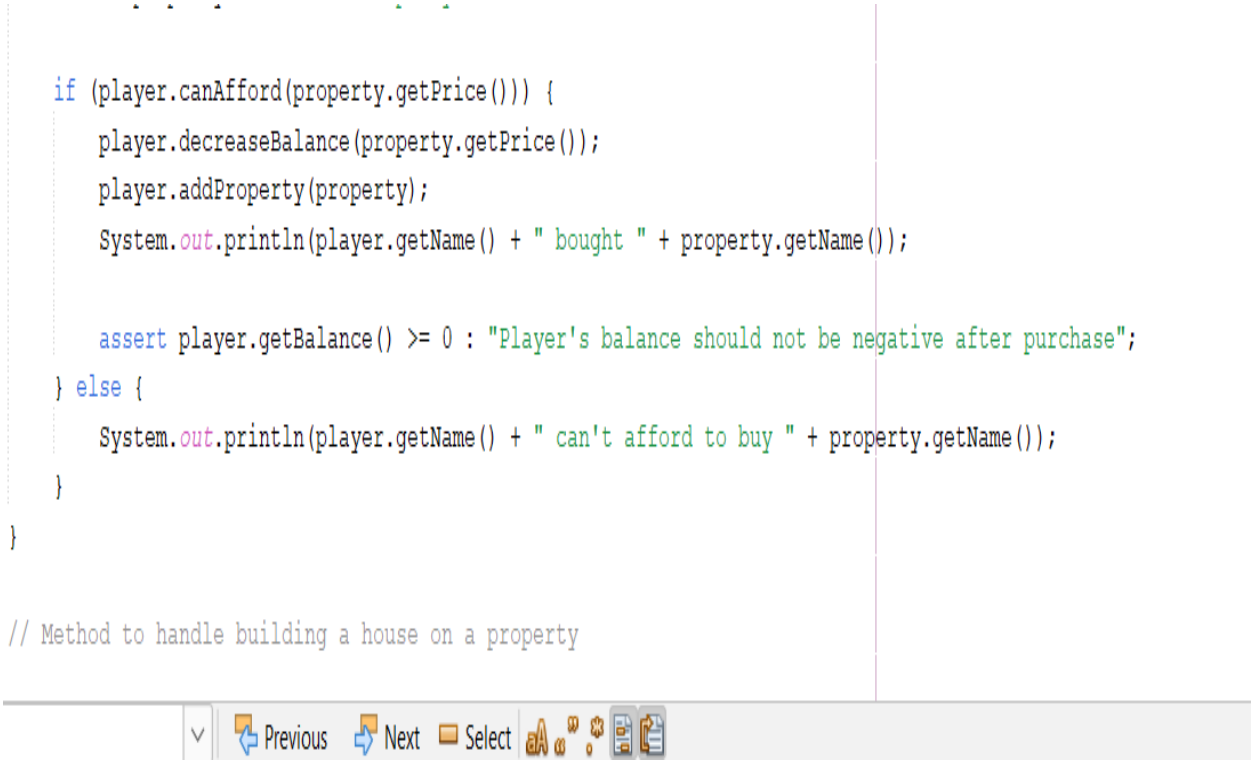
4 2 5

6 3 7

2 5 1

```
--- resources:3.3.1:resources (default-resources) @ Main ---
skip non existing resourceDirectory C:\Users\ASUS\OneDrive\Documents\NetBeansProjects\Main\src\main\resources

--- compiler:3.13.0:compile (default-compile) @ Main ---
Recompiling the module because of changed source code.
Compiling 12 source files with javac [debug release 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ Main ---
Invalid input: Dice roll must be between 1 and 6: 7
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.018 s
Finished at: 2024-10-09T06:15:42+02:00
------------------------------------------------------------------------
```

6 **Post-Condition Checks**: Assertions verify player's balance after transactions, maintaining internal consistency and preventing negative balances during financial operations.



```
    if (player.canAfford(property.getPrice())) {
        player.decreaseBalance(property.getPrice());
        player.addProperty(property);
        System.out.println(player.getName() + " bought " + property.getName());

        assert player.getBalance() >= 0 : "Player's balance should not be negative after purchase";
    } else {
        System.out.println(player.getName() + " can't afford to buy " + property.getName());
    }
}

// Method to handle building a house on a property
```


Black box Testing

1 **Undefined strategy in the file**: Validates that the system correctly handles unexpected input (undefined strategies) without needing to know the underlying implementation.

2 **FileNotFoundException**: Tests how the program reacts to the absence of an expected file, focusing on external conditions rather than internal code logic.

```
] --- resources:3.3.1:resources (default-resources) @ Main ---
- skip non existing resourceDirectory C:\Users\ASUS\OneDrive\Documents\NetBeansProjects\Main\src\main\resources

] --- compiler:3.13.0:compile (default-compile) @ Main ---
  Recompiling the module because of changed source code.
- Compiling 12 source files with javac [debug release 21] to target\classes

] --- exec:3.1.0:exec (default-cli) @ Main ---
- File not found: C:\Users\ASUS\OneDrive\Documents\NetBeansProjects\Main\src\main\java\com\mycompany\main\input7.txt
  ------------------------------------------------------------------------
  BUILD SUCCESS
  ------------------------------------------------------------------------
  Total time:  1.778 s
  Finished at: 2024-10-09T06:26:58+02:00
  ------------------------------------------------------------------------
```

3. **IOException**: Checks the program's ability to handle issues related to file reading, such as permissions or corrupted files.

.give any file where the permission is not givento the file for reading then you will get this Exception

4.**NumberFormatException**: Validates the handling of incorrect data formats from the file, ensuring the program behaves correctly when encountering unexpected input.

Input6.txt

The empty file

```
Compiling 12 source files with javac [debug release 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ Main ---
Invalid number format in the file: For input string: ""
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.902 s
Finished at: 2024-10-09T06:24:30+02:00
------------------------------------------------------------------------
```

5. **90**

6. **General Exception**: Catches any unanticipated errors, verifying that the program can handle unexpected situations without crashing.