EÖTVÖS LORÁND UNIVERSITY
INFORMATICS FACULTY
PROGRAMMING TECHNOLOGY COURSE
(Template belongs to Vilnius University)

Simplified Capitaly Game

Documentation of the Assignment No. 1, Task 3

Author: Monika Mirbakaite (YS0EG6)

ELTE email: ys0eg6@inf.elte.hu

Evaluated by: Lecturer Szendrei Rudolf,
Instructor Török Zoltán Ákos

Budapest
2024

1. DESCRIPTION

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 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).

2. ANALYSIS

2.1. Task Analysis (Key Points)

2.1.1. Game Elements

- There is a *board* that has *fields*.
- *Players* move around using a *dice*.
- A player can be *Greedy*, *Careful, and Tactical*.
- A field can be a *Property*, *Service*, or *Lucky field*.

2.1.2. Money

- If stepped on a **Property** player can buy it. Next time – they can build a house.
- Cost the *owner*. Just Property – 1000. House on Property – 4000.
- Cost for *another player*. No house – 500. House – 2000.

- If stepped on a **Service field** player pays *the bank* (the field parameter amount).
- If stepped on a **Lucky field**, the player gets some money (the field parameter amount).

2.1.3. Tactics

- **Greedy player**: Always buys.
- **Careful player:** Always buys only from the half of total balance if it can fit that budget.
- **Tactical player:** Always buys every other time.

2.1.4. When Out of Money

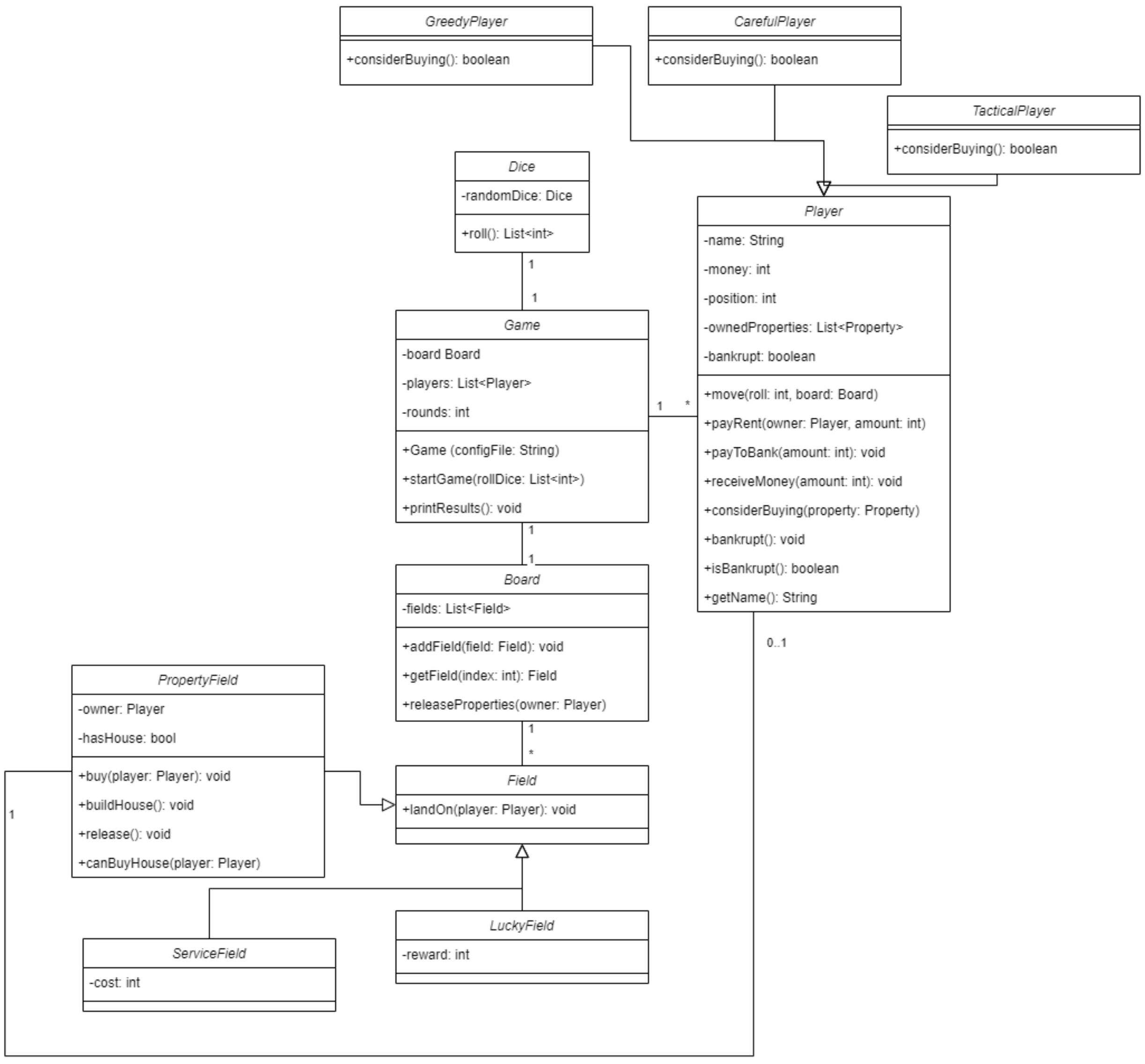
- Player that **has to pay, but runs out of money – loses**. Then, *owned properties are lost*, and become *free to buy*.

2.2. Solution Plan (Implementation Steps)

1. *Class diagram*. It will help to figure out the elements and how they interact with each other.
2. *Implementing the classes*. Define the classes with source code.
3. *Implementing the responsibilities in the game of each class*. Define what does what in source code.
4. *Implementing reading from files*. To be able to successfully start the game.
5. *Implementing needed testing*. To ensure that the program is working properly.

3. CLASS DIAGRAM

In Picture 1 a class diagram of Capitaly game I displayed. A Game has exactly one Board. A *Game* has one *Dice*. A *Board* contains multiple *Fields*. A *Property* field can be owned by zero or one *Player*. A game has multiple players. *PropertyField*, *ServiceField* and *LuckyField* inherit from the abstract class *Field* (also each one of them has their own additional methods) *GreedyPlayer, CarefulPlayer*, *TacticalPlayer* inherit from the abstract class *Player* (also each one of them has their own additional methods).


GreedyPlayer
+considerBuying(): boolean
CarefulPlayer
+considerBuying(): boolean
TacticalPlayer
+considerBuying(): boolean
Dice
-randomDice: Dice
+roll(): List<int>
1
1
Game
-board Board
-players: List<Player>
-rounds: int
+Game (configFile: String)
+startGame(rollDice: List<int>)
+printResults(): void
1
*
Player
-name: String
-money: int
-position: int
-ownedProperties: List<Property>
-bankrupt: boolean
+move(roll: int, board: Board)
+payRent(owner: Player, amount: int)
+payToBank(amount: int): void
+receiveMoney(amount: int): void
+considerBuying(property: Property)
+bankrupt(): void
+isBankrupt(): boolean
+getName(): String
0..1
1
1
Board
-fields: List<Field>
+addField(field: Field): void
+getField(index: int): Field
+releaseProperties(owner: Player)
1
*
PropertyField
-owner: Player
-hasHouse: bool
+buy(player: Player): void
+buildHouse(): void
+release(): void
+canBuyHouse(player: Player)
1
Field
+landOn(player: Player): void
ServiceField
-cost: int
LuckyField
-reward: int

Picture 1. Capitaly Game Class Diagram.

4. IMPLEMENTATION

4.1. Package “game”

4.1.1. Class of a Dice

```
package game;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;


public class Dice {
    private final List<Integer> rolls;
    private int currentRollIndex;
```

Constructor Dice is responsible for initiation of reading list of rolls from the file.

```
    public Dice(String filePath) throws IOException {
        this.rolls = new ArrayList<>();
        this.currentRollIndex = 0;
        readDiceRollsFromFile(filePath);
    }
```

Method readDiceRollsFromFile is responsible for reading list of rolls from the file.

```
    private void readDiceRollsFromFile(String filePath) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(filePath));
        String line;
        while ((line = br.readLine()) != null) {
            rolls.add(Integer.parseInt(line.trim()));
        }
        br.close();
    }
```

Method roll is responsible for getting values from the list.

```
public int roll() {
        if (currentRollIndex < rolls.size()) {
            return rolls.get(currentRollIndex++);
        } else {
            currentRollIndex = 0;
            return rolls.get(currentRollIndex++);
        }
    }
}
```

4.1.2. Class of a Game

```
package game;

import players.Player;
import java.io.*;
import java.util.*;

public class Game {
    private Board board;
    private List<Player> players;
    private int rounds;
    private Dice dice;
```

Constructor Game is responsible for reading input data of the game, adding according fields, players to the board

```
    public Game(String gameFile, String diceFile) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(gameFile));
        int numberOfFields = Integer.parseInt(br.readLine());
        board = new Board(numberOfFields);

        for (int i = 0; i < numberOfFields; i++) {
            String[] fieldInfo = br.readLine().split(" ");
            switch (fieldInfo[0]) {
                case "P":
                    board.addField(new PropertyField());
                    break;
                case "S":
                    int serviceCost = Integer.parseInt(fieldInfo[1]);
                    board.addField(new ServiceField(serviceCost));
                    break;
                case "L":
                    int luckyReward = Integer.parseInt(fieldInfo[1]);
                    board.addField(new LuckyField(luckyReward));
                    break;
            }
        }

        int numberOfPlayers = Integer.parseInt(br.readLine());
        players = new ArrayList<>();

        for (int i = 0; i < numberOfPlayers; i++) {
            String[] playerInfo = br.readLine().split(" ");
            String playerName = playerInfo[0];
            String strategy = playerInfo[1];
            switch (strategy) {
                case "Greedy":
                    players.add(new players.GreedyPlayer(playerName));
                    break;
                case "Careful":
                    players.add(new players.CarefulPlayer(playerName));
                    break;
                case "Tactical":
                    players.add(new players.TacticalPlayer(playerName));
                    break;
            }
        }

        rounds = Integer.parseInt(br.readLine());
        br.close();

        dice = new Dice(diceFile);
    }
```

Method startGame is responsible for continuing the game if there is rounds left, players moving around the board, and rolling a Dice by triggering roll method.

```
   public void startGame() {
        int currentRound = 0;
        while (currentRound < rounds && players.size() > 1) {
            for (int i = 0; i < players.size(); i++) {
                Player player = players.get(i);
                int roll = dice.roll();
                player.move(roll, board);

                if (player.isBankrupt()) {
                    System.out.println(player.getName() + " is bankrupt and removed
from the game.");
                    board.releaseProperties(player);
                    players.remove(i--);
                }
            }
            currentRound++;
        }

        printResults();
    }
```

Method printResults is responsible for printing out the results when the game is over.

```
private void printResults() {
        for (Player player : players) {
            System.out.println(player);
        }
    }
}
```

4.1.3. Class of a Board

```
package game;

import players.Player;
import java.util.*;

public class Board {
    public List<Field> fields;
```

Construct Board is responsible for determing the number of fields on the board.

```
    public Board(int numberOfFields) {
        fields = new ArrayList<>(numberOfFields);
    }
```

Method addField is responsible for adding fields on the board.

```
    public void addField(Field field) {
        fields.add(field);
    }
```

Method getField is responsible for accesing fields on the board.

```
    public Field getField(int index) {
        return fields.get(index % fields.size());
    }
```

Method releaseProperties is responsible for realeasing all the properties from the owner.

```
public void releaseProperties(Player owner) {
        for (Field field : fields) {
            if (field instanceof PropertyField) {
                PropertyField property = (PropertyField) field;
                if (property.getOwner() == owner) {
                    property.release();
                }
            }
        }
    }
}
```

4.1.4. Class of a Field

```
package game;

import players.Player;
```

```
public abstract class Field {
```

Abstract method is responsible to determine action when landed on a certain type of field. This will be extended in subclasses of Field.

```
      public abstract void landOn(Player player);
}
```

4.1.5. Class of a Property Field

```
package game;

import players.Player;

public class PropertyField extends Field {
    private Player owner;
    private boolean hasHouse;
```

Constructor PropertyField is responsible for creating a PropertyField instance with default values.

```
    public PropertyField() {
        owner = null;
        hasHouse = false;
    }
```

Method landOn is responsible for buying/paying rent when stepped on property.

```
    @Override
    public void landOn(Player player) {
        if (owner == null) {
            player.considerBuying(this);
        } else if (owner != player) {
            player.payRent(owner, hasHouse ? 2000 : 500);
        }
    }
```

Method canBuyHouse is responsible for checking whether a property has an owner, and whether it has a house.

```
    public boolean canBuyHouse(Player player) {
        return owner == player && !hasHouse;
    }
```

Method buildHouse is responsible for stating the fact that the house is built.

```
public void buildHouse() {
        hasHouse = true;
    }
```

Method getOwner is responsible for getting the owner of the property.

```
    public Player getOwner() {
        return owner;
    }
```

Method buy is responsible for setting the owner of the property.

```
public void buy(Player player) {
        owner = player;
    }
```

Method release is responsible for resetting the owner of the property to none and that it is free to buy, build house (because it is resetted to none).

```
    public void release() {
        owner = null;
        hasHouse = false;
    }
```

Method isOwned is responsible for checking if a property belongs to an owner.

```
public boolean isOwned() {
        return owner != null;
    }
}
```

4.1.6. Class of a Service Field

```
package game;

import players.Player;

public class ServiceField extends Field {
    private int cost;
```

Constructor ServiceField is responsible for initializing a specified cost amount.

```
    public ServiceField(int cost) {
        this.cost = cost;
    }
```

Method landOn is responsible for triggering the payBank method which deducts the cost by passing through a parameter.

```
    @Override
    public void landOn(Player player) {
        player.payBank(cost);
    }
}
```

4.1.7. Class of a Lucky Field

```
package game;

import players.Player;

public class LuckyField extends Field {
    private int reward;
```

Constructor LuckyField is responsible for initializing a specified reward amount.

```
    public LuckyField(int reward) {
        this.reward = reward;
    }
```

Method landOn is responsible for triggering the receiveMoney method which adds the reward by passing through a parameter.

```
    @Override
    public void landOn(Player player) {
        player.receiveMoney(reward);
    }
}
```

4.2. Package “players”

4.2.1. Class of Players

```
package players;

import game.Board;
import game.PropertyField;
import java.util.ArrayList;
import java.util.List;

public abstract class Player {
    protected String name;
    protected int money;
    protected int position;
    protected List<PropertyField> ownedProperties;
    protected boolean bankrupt;
```

Constructor Players is responsible for setting the initial data of the players.

```
    public Player(String name) {
        this.name = name;
        this.money = 10000;
        this.position = 0;
        this.ownedProperties = new ArrayList<>();
        this.bankrupt = false;
    }
```

Method move is responsible for updating a player's position on the game board.

```
    public void move(int roll, Board board) {
        position = (position + roll) % board.fields.size();
        board.getField(position).landOn(this);
    }
```

Method payRent is responsible for paying to the property owner. Player goes bankrupt if doen’t have enough money.

```
    public void payRent(Player owner, int amount) {
        if (money >= amount) {
            money -= amount;
            owner.receiveMoney(amount);
        } else {
            bankrupt();
        }
    }
```

Method payBank is responsible for paying to the bank. Player goes bankrupt if doen’t have enough money.

```
    public void payBank(int amount) {
        if (money >= amount) {
            money -= amount;
        } else {
            bankrupt();
        }
    }
```

Method receiveMoney is responsible for tranfering funds to the needed player.

```
    public void receiveMoney(int amount) {
        money += amount;
    }
```

Method considerBuying is responsible for the strategies of the players. Implementation will be found on subclasses of players.

```
    public void considerBuying(PropertyField property) {

    }
```

Method bankrupt is responsible for clearing out the properties of the player that lost the game.

```
    public void bankrupt() {
        bankrupt = true;
        ownedProperties.clear();
    }
```

Method isBankrupt is responsible for returning the bankrupt status of the player that lost the game.

```
    public boolean isBankrupt() {
        return bankrupt;
    }
```

Method getName is responsible for returning the name of the player that lost the game.

```
    public String getName() {
        return name;
    }
```

Method toString is responsible for returning the data of current player status in a string format.

```
    @Override
    public String toString() {
        return "Player: " + name + ", Money: " + money + ", Properties: " +
ownedProperties.size();
    }
}
```

4.2.2. Class of a Greedy Player

```
package players;

import game.PropertyField;

public class GreedyPlayer extends Player {
```

Constructor GreedyPlayer is responsible for calling the constructor of the superclass (Player) to initialize inherited data about the Greedy player.

```
public GreedyPlayer(String name) {
        super(name);
    }
```

Method considerBuying is responsible for implementing the game strategy of the Greedy player. Always buys if has enough money.

```
    @Override
    public void considerBuying(PropertyField property) {
        if (!property.isOwned() && money >= 1000) {
            property.buy(this);
            money -= 1000;
            ownedProperties.add(property);
        } else if (property.canBuyHouse(this) && money >= 4000) {
            property.buildHouse();
            money -= 4000;
        }
    }
}
```

4.2.3. Class of a Careful Player

```
package players;


import game.PropertyField;
```

Constructor CarefulPlayer is responsible for calling the constructor of the superclass (Player) to initialize inherited data about the Careful player.

```
public class CarefulPlayer extends Player {
    public CarefulPlayer(String name) {
        super(name);
    }
```

Method considerBuying is responsible for implementing the game strategy of the Careful player. Buys only if has enough money from half of their total funds.

```
    @Override
    public void considerBuying(PropertyField property) {
        if (!property.isOwned() && money >= 1000 && money / 2 >= 1000) {
            property.buy(this);
            money -= 1000;
            ownedProperties.add(property);
        } else if (property.canBuyHouse(this) && money / 2 >= 4000) {
            property.buildHouse();
            money -= 4000;
        }
    }
}
```

4.2.4. Class of a Tactical Player

```
package players;

import game.PropertyField;

public class TacticalPlayer extends Player {
    private boolean skipNextPurchase;
```

Constructor TacticalPlayer is responsible for calling the constructor of the superclass (Player) to initialize inherited data about the Tactical player. skipNextPurchase is false because player is currently considering buying.

```
    public TacticalPlayer(String name) {
        super(name);
        this.skipNextPurchase = false;
    }
```

Method considerBuying is responsible for implementing the game strategy of the Tactical player. If skipNextPurchase is true player stops the decision making of buying (sets it to false for another round). If it is false it continues it. Player buys if has enough money.

```
    @Override
    public void considerBuying(PropertyField property) {
        if (skipNextPurchase) {
            skipNextPurchase = false;
            return;
        }

        if (!property.isOwned() && money >= 1000) {
            property.buy(this);
            money -= 1000;
            ownedProperties.add(property);
        } else if (property.canBuyHouse(this) && money >= 4000) {
            property.buildHouse();
            money -= 4000;
        }

        skipNextPurchase = true;
    }
}
```

4.3. Class “Main”

```
import game.Game;

import java.io.IOException;

public class Main {
```

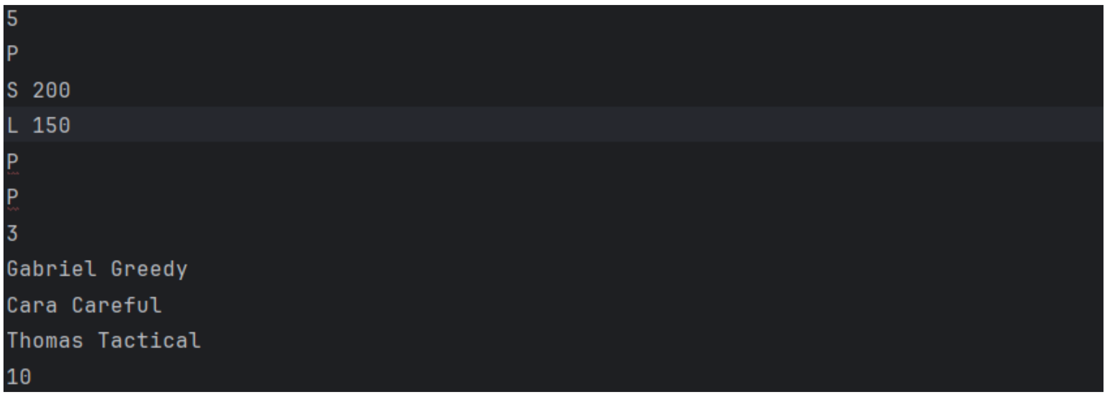
Method main is responsible for handling IO Exeptions with program files, initializing new game and starting it.

```
public static void main(String[] args) throws IOException {
        String configFilePath = "game.txt";
        String diceRollsFilePath = "dice.txt";

        Game game = new Game(configFilePath, diceRollsFilePath);
        game.startGame();
    }
}
```

4.4. Screenshots of Running Application

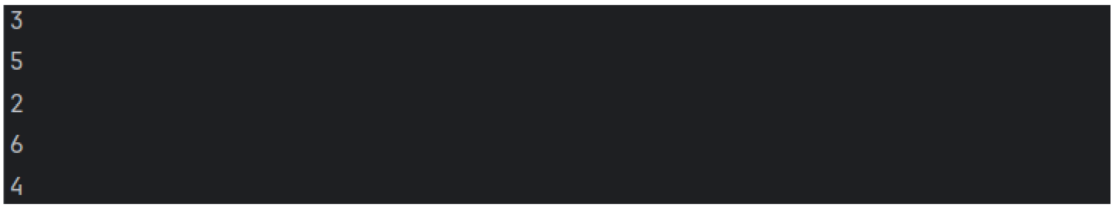
In Picture 2 the game information file is displayed. First line is number of fields. Then, their definition. Near the definitions, according to the type, cost of rent/reward. Then, the number of players. Then, their names and tactics. Then, number of rounds.



```
5
P
S 200
L 150
P
P
3
Gabriel Greedy
Cara Careful
Thomas Tactical
10
```

Picture 2. game.txt

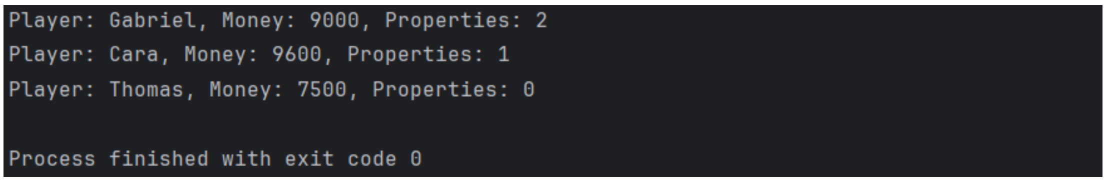
In Picture 3 dice roll information is displayed. It will iterate until reaches the bottom and then start from the top.



```
3
5
2
6
4
```

Picture 3. dice.txt

In Picture 4 the results after the end of the game are displayed.



```
Player: Gabriel, Money: 9000, Properties: 2
Player: Cara, Money: 9600, Properties: 1
Player: Thomas, Money: 7500, Properties: 0

Process finished with exit code 0
```

Picture 4. Output

5. TEST REPORT

5.1. Black Box

```
package test;
import game.*;
import players.*;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

public class WhiteBox {
```

This test will check if dice rolls are read correctly.

```
    @Test
    public void testDiceRollsFromFile() throws IOException {
        String filePath = "dice.txt";

        Dice dice = new Dice(filePath);

        assertEquals(3, dice.roll());
        assertEquals(5, dice.roll());
        assertEquals(2, dice.roll());
        assertEquals(6, dice.roll());
        assertEquals(4, dice.roll());
    }
```

This test will check if fields are accessed and wrapped correctly.

```
    @Test
    public void testFieldAccess() {
        Board board = new Board(5);
        Field field0 = new PropertyField();
        Field field1 = new ServiceField(200);
        Field field2 = new LuckyField(150);
        Field field3 = new PropertyField();
        Field field4 = new PropertyField();
        board.addField(field0);
        board.addField(field1);
        board.addField(field2);
        board.addField(field3);
        board.addField(field4);
        assertSame(field0, board.getField(0));
        assertSame(field1, board.getField(1));
        assertSame(field2, board.getField(2));
        assertSame(field3, board.getField(3));
        assertSame(field4, board.getField(4));
    }
```

This test will check if the game starts correctly.

```
    @Test
    public void testGameInitializationAndStart() throws IOException {
        // Initialize the game from the game.txt and dice.txt files
        Game game = new Game("game.txt", "dice.txt");
        game.startGame();
        assertTrue(true, "The game should initialize and complete successfully.");
    }
```

```
}
```

5.2. White Box

I had trouble creating test cases for black box testing because it is more complex.

6. USER STORIES

1. As a player, I want to read game data from a file so that I can start playing with already known settings.
2. As a player, I want to read dice data from a file so that I can start playing with already known settings.
3. As a player, I want to roll the dice to determine how many spaces I can move, so that I can navigate the game board.
4. As a player, I want to move my piece on the board based on my dice roll, so that I can land on different fields and interact with them.
5. As a player, I want to land on a property field and have the option to buy it, so that I can become richer.
6. As a player, I want to pay rent to another player if I land on their property, so that I can continue playing.
7. As a player, I want to see a list of properties I own, so that I can keep track.
8. As a player, I want to build houses on my owned properties if I have enough money, so that I can increase the rent value.
9. As a player, I want to be notified when I go bankrupt, so that I can understand the consequences of my financial decisions.
10. As a player, I want to implement a specific buying strategy (greedy, careful, tactical) when considering property purchases, so that I can play according to my preferred style.
11. As a player, I want to receive money when I land on a lucky field, so that I can improve my financial status in the game.
12. As a player, I want to release my properties if I go bankrupt, so that I can comply with the game rules.
13. As a player, I want to view the results at the end of the game, including who won and the final standings, so that I can evaluate my performance.
14. As a player, I want to manage my bank transactions (paying rent, receiving money, and paying off loans), so that I can keep track of my financial status.
15. As a player, I want to successfully start the game, so I would not have any issues while playing.