Programming Technology Documentation

Otar Jinchveladze

October 2024

Contents

1 Introduction

This document serves as the official documentation for the programming technology assignment. It outlines the goals and structure of developing as well as implementing the project. The primary objective of this assignment is to enhance our understanding of object-oriented programming principles. Additionally improve problem-solving skills through the design of a simulation-based game.

This document is created by Otar Jinchveladze, a second-year student pursuing computer science at Eötvös Loránd University (ELTE). This document will be reviewed by my teacher, Guettala Walid. He is a PhD student at ELTE with a background in deep learning and robotics, currently researching Graph Neural Networks.

Throughout this documentation, I will cover various aspects of the project. Including detailed descriptions of the exercise, the class diagram, and the implementation of methods. Moreover, testing procedures and results will be provided to demonstrate the functionality and reliability of the program. All this presents an opportunity to gain hands-on experience in designing modular, maintainable code and applying core programming concepts in a practical scenario.

2 Description of the Exercise

2.1 Exercise 3

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).

3 Class Diagram

The class diagram for the Capitaly game provides a comprehensive overview of the system architecture and the relationships between different components of the game. It is designed to illustrate the key classes, their attributes, methods, and the interactions among them.

Classes: Player (Abstract Class), Greedy, Careful, Tactical (Subclasses of Player). Field (Abstract Class), Property, Service, Lucky (Subclasses of Field). Bank. Board. There is a Main class in the program as well responsible the file reading.

3.1 Main Class Diagram

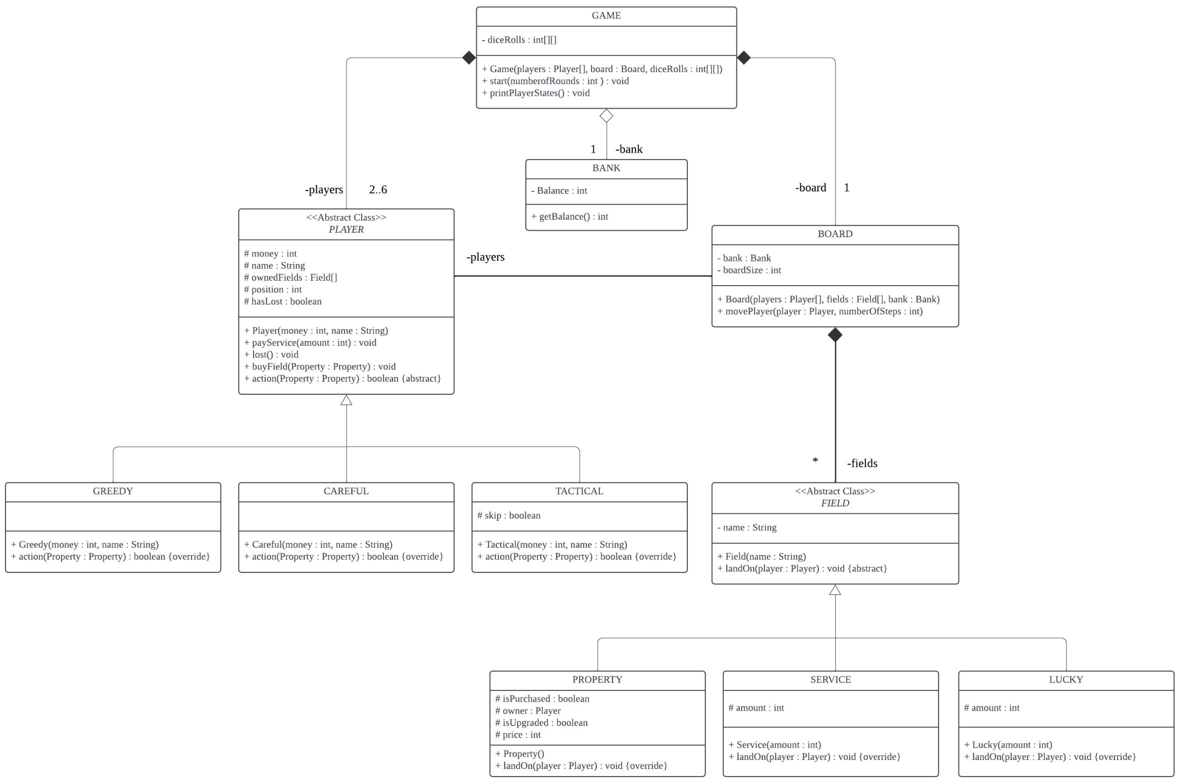

GAME
- diceRolls : int[][]
+ Game(players : Player[], board : Board, diceRolls : int[][])
+ start(numberofRounds : int) : void
+ printPlayerStates() : void
1
-bank
BANK
- Balance : int
+ getBalance() : int
-players
2..6
-board
1
<<Abstract Class>>
PLAYER
# money : int
# name : String
# ownedFields : Field[]
# position : int
# hasLost : boolean
+ Player(money : int, name : String)
+ payService(amount : int) : void
+ lost() : void
+ buyField(Property : Property) : void
+ action(Property : Property) : boolean {abstract}
-players
BOARD
- bank : Bank
- boardSize : int
+ Board(players : Player[], fields : Field[], bank : Bank)
+ movePlayer(player : Player, numberOfSteps : int)
*
-fields
GREEDY
+ Greedy(money : int, name : String)
+ action(Property : Property) : boolean {override}
CAREFUL
+ Careful(money : int, name : String)
+ action(Property : Property) : boolean {override}
TACTICAL
# skip : boolean
+ Tactical(money : int, name : String)
+ action(Property : Property) : boolean {override}
<<Abstract Class>>
FIELD
- name : String
+ Field(name : String)
+ landOn(player : Player) : void {abstract}
PROPERTY
# isPurchased : boolean
# owner : Player
# isUpgraded : boolean
# price : int
+ Property()
+ landOn(player : Player) : void {override}
SERVICE
# amount : int
+ Service(amount : int)
+ landOn(player : Player) : void {override}
LUCKY
# amount : int
+ Lucky(amount : int)
+ landOn(player : Player) : void {override}

Figure 1: Class Diagram for the Capitaly Game

3.2 Part One

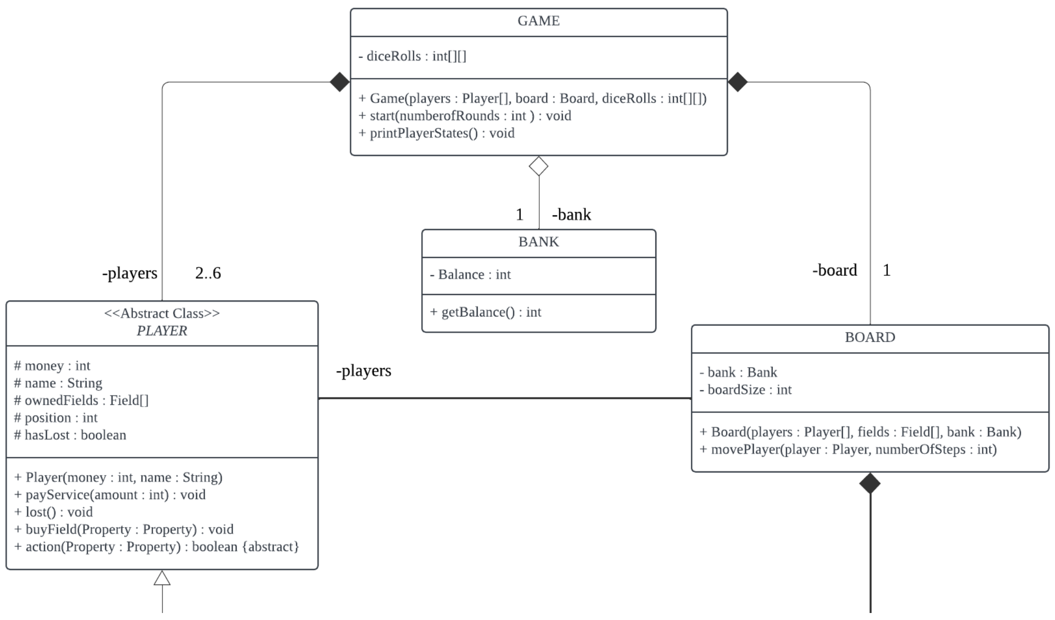

GAME
- diceRolls : int[][]
+ Game(players : Player[], board : Board, diceRolls : int[][])
+ start(numberofRounds : int) : void
+ printPlayerStates() : void
1
-bank
BANK
- Balance : int
+ getBalance() : int
-players
2..6
-board
1
<<Abstract Class>>
PLAYER
# money : int
# name : String
# ownedFields : Field[]
# position : int
# hasLost : boolean
+ Player(money : int, name : String)
+ payService(amount : int) : void
+ lost() : void
+ buyField(Property : Property) : void
+ action(Property : Property) : boolean {abstract}
-players
BOARD
- bank : Bank
- boardSize : int
+ Board(players : Player[], fields : Field[], bank : Bank)
+ movePlayer(player : Player, numberOfSteps : int)

Figure 2: Class Diagram First Part Zoomed

3.3 Part Two

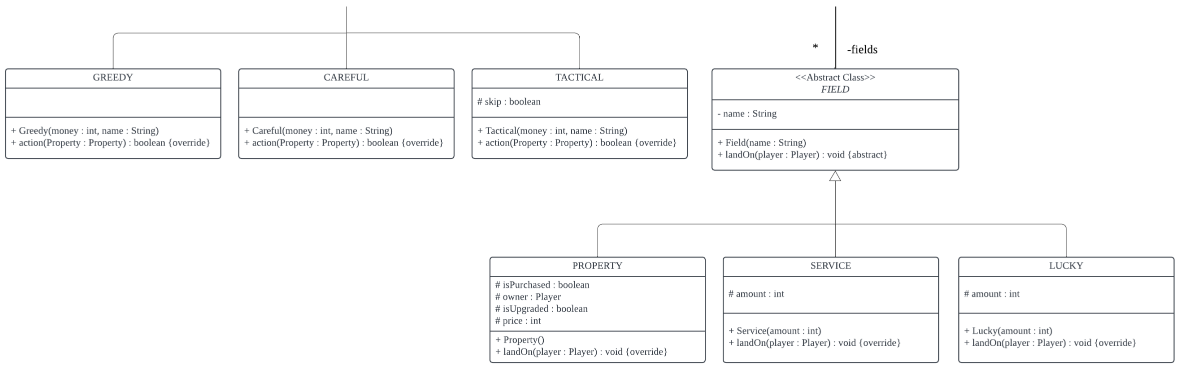

GREEDY
+ Greedy(money : int, name : String)
+ action(Property : Property) : boolean {override}
CAREFUL
+ Careful(money : int, name : String)
+ action(Property : Property) : boolean {override}
TACTICAL
# skip : boolean
+ Tactical(money : int, name : String)
+ action(Property : Property) : boolean {override}
*
-fields
<<Abstract Class>>
FIELD
- name : String
+ Field(name : String)
+ landOn(player : Player) : void {abstract}
PROPERTY
# isPurchased : boolean
# owner : Player
# isUpgraded : boolean
# price : int
+ Property()
+ landOn(player : Player) : void {override}
SERVICE
# amount : int
+ Service(amount : int)
+ landOn(player : Player) : void {override}
LUCKY
# amount : int
+ Lucky(amount : int)
+ landOn(player : Player) : void {override}

Figure 3: Class Diagram Second Part Zoomed

4 Description of Methods

This documentation will not include getter and setter methods, as they are straightforward and commonly understood for managing access to private and protected fields. Getters and setters serve a clear purpose: getters allow external classes to access the values of private fields, while setters enable them to modify those values. Since these methods are standard practices in object-oriented programming and their functionality is generally familiar to developers, we will focus on documenting the more complex methods and functionality of the classes. This approach helps streamline the documentation by omitting repetitive explanations and allowing us to concentrate on the unique features and behaviors of the system that may require further clarification or context.

4.1 Game Class

4.1.1 Method: start(int n)

Initiates the game for a specified number of rounds. It iterates over the players and their respective dice rolls, moving them around the board and calling the landOn() method of the fields they land on. The method returns void and takes NumberOfRounds as a parameter which is an integer number.

4.1.2 Method: printPlayerStates()

Displays the current state of each player, including their name, money, number of owned properties, and their position on the board. This is the method used for our output which is printing balance and number of owned properties.

4.2 Player Class

4.2.1 Method: payService(int a)

Deducts a specified amount from the player's balance, typically used when landing on a service field. Method's return type is void and it takes amount as a parameter which is an integer number.

4.2.2 Method: lost()

Handles the logic when a player loses, resetting owned properties and clearing the list of owned fields. Method's return type is void.

4.2.3 Method: buyField(Property p)

Adds a specified property to the player's list of owned fields. Method's return type is void takes p type of Property as a parameter.

4.2.4 Method: action(Property p)

An abstract method for subclasses to define their strategy for buying properties based on their specific behaviors. Method's return type is boolean and takes p type of Property as a parameter.

4.3 Player Subclasses (Greedy, Careful, Tactical)

4.3.1 Method: action(Property p) (Greedy)

Override: Buys if the player has more money than the property's price.

4.3.2 Method: action(Property p) (Careful)

Override: Buys if the player has at least half the property's price.

4.3.3 Method: action(Property p) (Tactical)

Override: Alternates between buying and not buying each round.

4.4 Field Class

4.4.1 Method: landOn(Player p)

An abstract method that must be implemented by subclasses to define what happens when a player lands on that field.

4.5 Field Subclasses

4.5.1 Method: landOn(Player p) (Property)

Override: Handles the logic when a player lands on a property, including purchasing it, paying rent, and upgrading it.

4.5.2 Method: landOn(Player p) (Service)

Override: Deducts a specified service cost from the player when they land on a service field.

4.5.3 Method: landOn(Player p) (Lucky)

Override: Adds a specified amount of money to the player's balance when they land on a lucky field.

4.6 Bank Class

4.6.1 Method: getBalance()

Returns the current balance of the bank, which is initialized to a default value.

4.7 Board Class

4.7.1 Method: movePlayer(Player p, int n)

Moves the player by a specified number of steps around the board, calculates the new position and calls the landOn() method for the field the player lands on.

4.8 InvalidDataException Class

4.8.1 Method: InvalidDataException(String s)

A custom exception that takes a message and prints it, used to handle invalid data input when reading from the configuration file.

5 Testing

In this part of the documentation, I will use the **Corner Case method** to test how our application performs in unusual or extreme situations. This involves checking the system with inputs that are at the edges of acceptable limits or under rare conditions. By focusing on these corner cases, we aim to uncover potential problems that regular testing might miss, ensuring the application works reliably in different scenarios and enhancing user confidence in its performance.

5.1 Corner Cases:

5.1.1 LandsOnOwnUpgradedProperty()

Description: Simulates a player landing on their own property that has already been upgraded. The player should not lose or gain money by landing on their own property.
Expected Output: The player's money should remain the same after landing on their upgraded property.
Scenario: Tests that the game does not incorrectly penalize or reward players for landing on their own properties.

5.1.2 BuyPropertyWithExactMoney()

Description: Tests a scenario where a player has exactly enough money to purchase a property. The test verifies that player is allowed to buy the property however he loses the game.
Expected Output: The player's money should become 0, property ownership should get cleared.
Scenario: Ensures that players lose after purchasing properties when their money is exactly equal to the property price.
Addition: There might be different ways to write this program. In this case I made sure that player with 0 money does not lose simply by making condition less and not less or equal.

5.1.3 BankruptcyWithMultipleProperties()

Description: Simulates a scenario where a player buys multiple properties and then goes bankrupt by paying a service fee that exceeds their remaining money.
Expected Output: The player should lose ownership of all properties after bankruptcy, and the properties should have no owners.
Scenario: Tests the game's handling of bankruptcy and whether the player's properties are forfeited correctly when they lose all their money.

5.1.4 ServiceFieldExactFee()

Description: Tests if a player can land on a service field and pay the exact service fee, leaving them with 0 money which leads to losing a game.
Expected Output: The player's money should become 0 after paying the service fee and player should lose the game.
Scenario: Ensures that service fees are deducted correctly, especially when the player's money is exactly equal to the service fee and player loses the game accordingly.

5.1.5 TacticalPlayerSkipsTurn()

Description: Tests the behavior of a Tactical player, who skips buying properties every other turn. The player buys the first property but skips the second.
Expected Output: After the first turn, the player should own one property. After the second turn (where the player skips), the number of owned properties should remain the same.
Scenario: Verifies that Tactical players correctly skip buying properties according to the game rules.

5.2 White Box Testing:

5.2.1 LandWithExactMoney

Description: Checking whether a player can successfully buy a property when they have exact amount of money. The player should have 0 money left after purchasing the property consequently he should lose the game.

5.2.2 ServicePayment

Description: Validates whether a player correctly pays the service fee when landing on a service field or not. The player's money should decrease by the Service amount. The player should not lose the game as long as they have some money left after payment.

5.2.3 BankruptcyPlayer

Description: Verifies that a player goes bankrupt and loses all their properties when they land on a service field with a fee that exceeds their current money. The player should lose and their properties should be removed.

5.2.4 buyingProperty

Description: Ensures that when a player lands on a property they buy it. The player's money should decrease by the property's price *(1000)*, and the property should be marked as owned by the player.

5.2.5 landingOnLucky

Description: Checks that when a player lands on a Lucky field, they receive the bonus. The player's money should increase by the bonus amount.

5.3 Black Box Testing:

5.3.1 notEnoughMoney

Description: Test a player's attempt to buy a property when they don't have enough money.

5.3.2 HigherService

Description: Test a scenario where a player lands on a service field and needs to pay a fee that exceeds their available money, causing bankruptcy.

5.3.3 IncorrectFilePath

Description: Test a scenario where the incorrect file path is given to a program.
Expected Output: Error reading file: ”given filepath”.

6 Special Format of Testing:

1. **LandsOnOwnUpgradedProperty**

- **AS A** player
- **I WANT TO** land on my own upgraded property
- **GIVEN** I own an upgraded property
- **WHEN** I land on it
- **THEN** my money should remain the same after the action.

2. **BuyPropertyWithExactMoney**

- **AS A** player
- **I WANT TO** buy a property when I have exactly enough money
- **GIVEN** I have an amount equal to the property price
- **WHEN** I make the purchase
- **THEN** my money should become 0, and I should lose the game.

3. **BankruptcyWithMultipleProperties**

- **AS A** player
- **I WANT TO** go bankrupt after paying a service fee that exceeds my money
- **GIVEN** I own multiple properties and have insufficient funds
- **WHEN** I pay the service fee
- **THEN** I should lose ownership of all my properties, and they should have no owners.

4. **ServiceFieldExactFee**

- **AS A** player
- **I WANT TO** land on a service field and pay the exact fee
- **GIVEN** my available money is equal to the service fee
- **WHEN** I make the payment
- **THEN** my money should become 0, and I should lose the game.

5. TacticalPlayerSkipsTurn

- **AS A** Tactical player
- **I WANT TO** skip buying properties every other turn
- **GIVEN** I buy the first property on my first turn
- **WHEN** I skip buying on the second turn
- **THEN** my number of owned properties should remain the same.

6. LandWithExactMoney

- **AS A** player
- **I WANT TO** buy a property with my exact amount of money
- **GIVEN** I have just enough money for the purchase
- **WHEN** I buy the property
- **THEN** I should have 0 money left and subsequently lose the game.

7. ServicePayment

- **AS A** player
- **I WANT TO** ensure I can correctly pay the service fee
- **GIVEN** I land on a service field
- **WHEN** I pay the fee
- **THEN** my money should decrease by the service amount, and I should not lose the game unless my funds reach 0.

8. BankruptcyPlayer

- **AS A** player
- **I WANT TO** lose all my properties upon bankruptcy
- **GIVEN** I land on a service field with a fee greater than my current money
- **WHEN** I cannot pay the fee
- **THEN** I should go bankrupt, and all my properties should be removed.

9. BuyingProperty

- **AS A** player
- **I WANT TO** buy a property when I land on it
- **GIVEN** I have sufficient funds
- **WHEN** I make the purchase

- **THEN** my money should decrease by the property price (e.g., 1000), and I should be marked as the property owner.

10. LandingOnLucky

- **AS A** player
- **I WANT TO** receive a bonus when landing on a Lucky field
- **GIVEN** I land on the Lucky field
- **WHEN** the bonus is applied
- **THEN** my money should increase by the bonus amount.

11. NotEnoughMoney

- **AS A** player
- **I WANT TO** attempt to buy a property when I don't have enough money
- **GIVEN** my current balance is less than the property price
- **WHEN** I try to make the purchase
- **THEN** I should receive a notification that I can't afford the property.

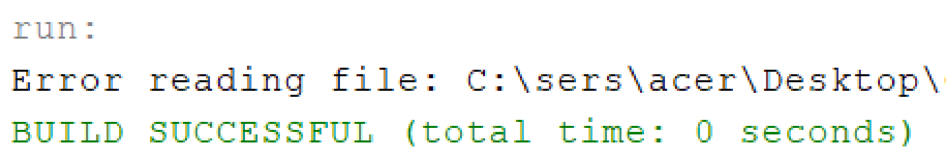
12. HigherService

- **AS A** player
- **I WANT TO** land on a service field and face bankruptcy
- **GIVEN** the service fee exceeds my available money
- **WHEN** I try to pay the fee
- **THEN** I should go bankrupt.

6.1 Screenshots

13. IncorrectFilePath

- **AS A** user
- **I WANT TO** provide an incorrect file path to the program
- **GIVEN** I attempt to run the program with a wrong file location
- **WHEN** the file is accessed
- **THEN** I should see the message: ”Error reading file: given filepath.”

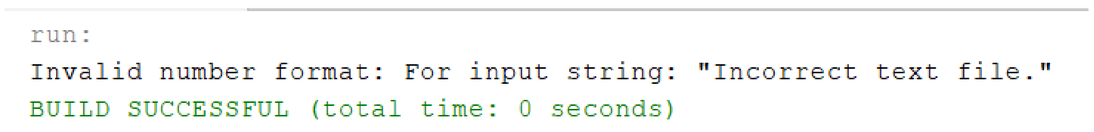


```
run:
Error reading file: C:\sers\acer\Desktop\
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 4: Screenshot of error message for incorrect file path.

14. IncorrectFileText

- **AS A** user
- **I WANT TO** test the program with an incorrectly formatted text file
- **GIVEN** I provide a text file with invalid contents
- **WHEN** the program attempts to read it
- **THEN** I should see the message: ”Invalid number format: For input string: ’Incorrect text file.’”



```
run:
Invalid number format: For input string: "Incorrect text file."
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 5: Screenshot of error message for incorrect file text.