Capitaly Game

Andrei Chubenko

September 2025

# Introduction

This document serves as the official documentation for the programming technology assignment. It outlines the goals and structure of development 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 modeling of a real-life interaction.

# Description of the Exercise

## Exercise 4

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)**

# Class Diagram

The class diagram for the Capitoly provides a comprehensive overview of the system architecture and the relationships between different components .

**Classes:** AbstractTile, Dice, GameCycle Main(MainClass containing point of entry function), BuyingProcess, GameConfig, Lucky, Property, PropertyState, Service, Strategy(enum), TileFactory, TileType(enum)

## Main Class Diagram

A white sheet of paper with text

AI-generated content may be incorrect.

# 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.

## Main Class

### Method: main()

Entry point of the program that reads game file, and validates all of the inputs with validation functions like: **validateTileCount**(int tiles),

**validateGameBoardSize**(ArrayList<AbstractTile> gameBoard, int expectedTiles),

**validatePlayersCount**(ArrayList<Player> players),

**validateTileType**(String tileType),

**validateTileValue**(String tileType, int value),

**validatePlayerName**(String playerName, Set<String> existingNames),

**validatePlayerStrategy**(String strategy),

**validateDiceValue**(short diceValue),

**validateFinalGameState**(ArrayList<AbstractTile> gameBoard, ArrayList<Player> players, int expectedTiles)

### Methods for processing input

There are three methods for processing input:

processPlayerLine(String line, ArrayList<Player> players, Set<String> playerNames),

processTileLine(String line, ArrayList<AbstractTile> gameBoard),

processDiceRollLine(String line, ArrayList<Short> diceRolls)

All of them are relying on flags to properly process inputs and throw out appropriate errors when there issue with input files

### Method: runGame(String gameFile)

### This method is responsible for running the whole game, reads data from the config file and it reads file line by line.

### First of all it reads all of the inputs, validates them,

### Then Based on the inputs it decides whether it will use predefined inputs or random inputs generated in the dice, then it launches the most important class where everything is happening GameCycle()

## AbstractTile

## Method: getMoney()

## This method by default returns 0 and by default is the designed to be overwritten by it’s child classes

## TileFactory

## createTile(String type, int amount)

## The createTile method implements the Factory design pattern to create appropriate tile objects based on the specified type string

## GameCycle

## printGameBoard(ArrayList<AbstractTile> gameBoard)

## Method responsible for printing out the board to make it obvious what happens in the game

## GameCycle(ArrayList<Player> players, ArrayList<AbstractTile> GameBoard)

## This is where the main loop is located and all of logic is handled . It initializes all players at position 0, then runs rounds until either one player remains or 100 rounds are completed. All players follow strict predefined behavior.

## 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.

## Also players can buy properties for 400o dollars(the amount can be modified in config file)

## Game is over when all players except one lost their money, or 100 rounds is reached

## eliminatePlayer(Player player, ArrayList<AbstractTile> GameBoard)

## The eliminatePlayer method handles the removal of a bankrupt player from the game. It prints the player's elimination, iterates through all tiles on the board to find properties owned by the eliminated player, resets those properties to unowned and empty state, and sets the player's money to -1 to mark them as eliminated

## countProperties(Player player, ArrayList<AbstractTile> GameBoard)

## This method counts all of the properties owned by every player

## printPlayerProperties(Player player, ArrayList<AbstractTile> GameBoard)

## This method prints all of properties owned by specific person

## Dice

## setDiceRolls

## This method either uses predefined dicerolls and sets them into an array

## ThrowDice(int position, int GameBoardSize)

## The ThrowDice method generates a dice roll and calculates the new position on the game board. There is a safeguard so player won’t generate a value that will put player outside game board

## Player

## Player(String Name, String strategy)

## Player constructor initializes a new player with the specified name and strategy

## AddMoney(int money)

## AddMoney method increases the player's current money balance by the specified amount

## SubstractMoney()

## AddMoney method increases the player's current money balance by the specified amount

## BuyingDesicion(AbstractTile tile)

## The BuyingDesicion method determines whether the player should purchase a property or build a house based on their strategy.

## toString()

## The toString method creates a formatted string representation of the player object.

## PropertyState Enum

## Default options for this enum: EMPTY, HOUSE

## 4.8 Strategy Enum

## Default options for this enum: GREEDY, CAREFUL, TACTICAL

## 4.9 TileType Enum

## Default options for this enum: SERVICE, PROPERTY, LUCKY

# Testing

# Test Cases

**Player Tests**

**1. Player Initial Money (Blackbox)**

* **As a:** Capitaly game player
* **I want to:** Start with the correct initial amount of money
* **Given:** A new player is created with any strategy
* **When:** I check the player's money balance
* **Then:** The player should have exactly 10,000 money units

**2. Add Money to Player (Blackbox)**

* **As a:** Capitaly game player
* **I want to:** Increase my money when I receive payments
* **Given:** A player with an existing money balance
* **When:** I call AddMoney() with a positive amount
* **Then:** The player's balance increases by the specified amount

**3. Subtract Money from Player (Blackbox)**

* **As a:** Capitaly game player
* **I want to:** Decrease my money when I make payments
* **Given:** A player with sufficient money balance
* **When:** I call SubtractMoney() with a positive amount
* **Then:** The player's balance decreases by the specified amount

**4. Prevent Negative Money Addition (Blackbox)**

* **As a:** Capitaly game system
* **I want to:** Prevent invalid money operations
* **Given:** A player with any money balance
* **When:** I attempt to add negative money using AddMoney()
* **Then:** An IllegalArgumentException should be thrown

**5. Greedy Player Buying Decision (WhiteBox)**

* **As a:** Greedy strategy player
* **I want to:** Buy any property I can afford
* **Given:** An unowned property and sufficient money (≥ 1000)
* **When:** I call BuyingDecision() on the property
* **Then:** The method returns true (buy the property)

**6. Careful Player Buying Decision (Whitebox)**

* **As a:** Careful strategy player
* **I want to:** Only buy properties when I have enough reserve money
* **Given:** An unowned property and less than 2000 money
* **When:** I call BuyingDecision() on the property
* **Then:** The method returns false (don’t buy the property)

**7. Tactical Player Skip Pattern (Whitebox)**

* **As a:** Tactical strategy player
* **I want to:** Skip every second buying opportunity
* **Given:** Two consecutive buying opportunities for properties
* **When:** I call BuyingDecision() twice
* **Then:** The first call returns true and the second returns false

**Dice Tests**

**1. Use Predefined Dice Rolls (Whitebox)**

* **As a:** Game developer testing the system
* **I want to:** Use predetermined dice rolls for consistent testing
* **Given:** A list of predefined dice rolls [3, 5]
* **When:** I call ThrowDice() twice after setting the rolls
* **Then:** The method returns 3 for the first call and 5 for the second call

**2. Handle Empty Dice Roll List (Whitebox)**

* **As a:** Game system
* **I want to:** Fall back to random generation when no predefined rolls exist
* **Given:** An empty list of dice rolls
* **When:** I call setDiceRolls() with the empty list and then ThrowDice()
* **Then:** The system should use random dice generation

**3. Exhaust Predefined Rolls (Whitebox)**

* **As a:** Game system
* **I want to:** Switch to random generation after using all predefined rolls
* **Given:** A list with only one predefined roll [2]
* **When:** I call ThrowDice() twice
* **Then:** The first call returns 2, and the second uses random generation

**4. Dice Position Wrapping (Whitebox)**

* **As a:** Game system
* **I want to:** Ensure players wrap around the board correctly
* **Given:** A player at position 8 on a 10-tile board with dice roll 5
* **When:** I call ThrowDice(8, 10)
* **Then:** The new position should be 3 (wrapping around the boarder

**Player Behavior tests (Whitebox)**

* **Greedy Player Insufficient Funds**
  + **As a:**Greedy strategy player
  + **I want to:**Not buy properties I cannot afford
  + **Given:**An unowned property and insufficient money (< 1000)
  + **When:**I call BuyingDecision() on the property
  + **Then:**The method returns false (cannot afford the property)
* **Greedy Player House Upgrade - Sufficient Funds**
  + **As a:**Greedy strategy player
  + **I want to:**Upgrade my empty properties to houses when I can afford it
  + **Given:**A property I own with EMPTY state and sufficient money (≥ 4000)
  + **When:**I call BuyingDecision() on my property
  + **Then:**The method returns true (upgrade to house)

1. **Greedy Player House Upgrade - Insufficient Funds**
   * **As a:**Greedy strategy player
   * **I want to:**Not upgrade properties when I cannot afford house costs
   * **Given:**A property I own with EMPTY state and insufficient money (< 4000)
   * **When:**I call BuyingDecision() on my property
   * **Then:**The method returns false (cannot afford house upgrade)
2. **Greedy Player Property with House**
   * **As a:**Greedy strategy player
   * **I want to:**Not attempt to upgrade properties that already have houses
   * **Given:**A property I own that already has a HOUSE
   * **When:**I call BuyingDecision() on my property
   * **Then:**The method returns false (no further upgrades possible)
3. **Greedy Player Non-Property Tile**
   * **As a:**Greedy strategy player
   * **I want to:**Not attempt buying decisions on non-property tiles
   * **Given:**A Service tile
   * **When:**I call BuyingDecision() on the service tile
   * **Then:**The method returns false (cannot buy services)
4. **Careful Player Sufficient Funds for Purchase**
   * **As a:**Careful strategy player
   * **I want to:**Buy properties when I have sufficient reserve money
   * **Given:**An unowned property and money ≥ 2000
   * **When:**I call BuyingDecision() on the property
   * **Then:**The method returns true (safe to buy property)
5. **Careful Player House Upgrade - Sufficient Funds**
   * **As a:**Careful strategy player
   * **I want to:**Upgrade my properties to houses when I have enough reserve money
   * **Given:**A property I own with EMPTY state and money ≥ 8000
   * **When:**I call BuyingDecision() on my property
   * **Then:**The method returns true (safe to upgrade to house)
6. **Careful Player House Upgrade - Insufficient Funds**
   * **As a:**Careful strategy player
   * **I want to:**Not risk upgrading properties without sufficient reserves
   * **Given:**A property I own with EMPTY state and money < 8000
   * **When:**I call BuyingDecision() on my property
   * **Then:**The method returns false (not enough reserve money)
7. **Careful Player Property with House**
   * **As a:**Careful strategy player
   * **I want to:**Not attempt to upgrade properties that already have houses
   * **Given:**A property I own that already has a HOUSE
   * **When:**I call BuyingDecision() on my property
   * **Then:**The method returns false (no further upgrades possible)
8. **Careful Player Non-Property Tile**
   * **As a:**Careful strategy player
   * **I want to:**Not attempt buying decisions on non-property tiles
   * **Given:**A Service tile
   * **When:**I call BuyingDecision() on the service tile
   * **Then:**The method returns false (cannot buy services)
9. **Tactical Player Odd Counter Purchase**
   * **As a:**Tactical strategy player
   * **I want to:**Consider buying on odd-numbered opportunities when I can afford it
   * **Given:**Counter at 0 (next call = 1, odd), unowned property, and sufficientmoney
   * **When:**I call BuyingDecision() on the property
   * **Then:**The method returns true and counter increments to 1
10. **Tactical Player Even Counter Skip**
    * **As a:**Tactical strategy player
    * **I want to:**Always skip even-numbered opportunities regardless of affordability
    * **Given:**Counter at 1 (next call = 2, even), unowned property, and sufficient money
    * **When:**I call BuyingDecision() on the property
    * **Then:**The method returns false and counter increments to 2
11. **Tactical Player Odd Counter Insufficient Funds**
    * **As a:**Tactical strategy player
    * **I want to:**Not buy properties I cannot afford, even on odd-numbered opportunities
    * **Given:**Counter at 0 (next call = 1, odd), unowned property, and insufficient money
    * **When:**I call BuyingDecision() on the property
    * **Then:**The method returns false and counter increments to 1
12. **Tactical Player House Upgrade - Odd Counter**
    * **As a:**Tactical strategy player
    * **I want to:**Consider house upgrades on odd-numbered opportunities when affordable
    * **Given:**Counter at 0 (next call = 1, odd), owned property with EMPTY state, sufficient money
    * **When:**I call BuyingDecision() on my property
    * **Then:**The method returns true and counter increments to 1
13. **Tactical Player House Upgrade - Insufficient Funds**
    * **As a:**Tactical strategy player
    * **I want to:**Not upgrade properties I cannot afford, even on odd opportunities
    * **Given:**Counter at 0 (next call = 1, odd), owned property with EMPTY state, insufficient money
    * **When:**I call BuyingDecision() on my property
    * **Then:**The method returns false and counter increments to 1
14. **Tactical Player Non-Property Tile**
    * **As a:**Tactical strategy player
    * **I want to:**Not increment counter for non-property tiles
    * **Given:**A Service tile
    * **When:**I call BuyingDecision() on the service tile
    * **Then:**The method returns false and counter remains unchanged