Monopoly Game Simulator

Software Requirements Specification

V1.3

Lead Software Engineers

Muhammed Bera KOÇ

Salih ÖZYURT

Mücahit TANACIOĞLU

Ahmet LEKESİZ

CUSTOMERS

Murat Can GANİZ

Serap KORKMAZ

Content

1. Introduction

1.1 Purpose

The purpose of project provides customers to observer famous Monopoly Game in simulation.

1.2 Scope

Monopoly Game is played in simulation by simulated players(SP) that the customers define count of. Monopoly Game will teach customers how to play the game if he/she doesn't know play game. And it gives a manifest chance to create new strategies about the game. The game heavily depends on random events. Actually all choice mechanisms in the game created based on random values generated by the computer itself. Game has some limits such as number of players must be between 2 and 8. There are 6 different types of squares. Squares are blocks which creates the game board. Tax square takes money from the SP whenever SP lands on it.
Go square is a square which has no interaction yet whenever SP start a new turn on the board -by that it is meant that SP must traverse all the board.- go square(starting square) gives the user a specified amount of money by the customer to the SP. The regular square on the other hand has nothing to do. The jail square captures the player and does not let him go until one of the two conditions is satisfied. Either player rolls two dice with same faces or after three roll change he shall pay a predestined amount of money. The Property square on the other hand is a square which holds a building to be bought or to be paid. If a building has an owner, then the owner will take some money from the SP who has trodden on that land. Otherwise an unowned property can be bought by an SP if SP satisfies the condition of buying. The last type of square is Luck squares. They have dozens of different types. Each kind of luck card has a specific behaviour. They can be bountiful or wicked. This all depends on player’s chance. There is a bankruptcy case in the game which occurs when SP balance is less than zero. Game ends when other than the one player, all players are bankrupt.

1.3 Definitions, Acronyms, and Abbreviations

SP(Simulated Player): A simulated player is a player which is controlled by the system - or program-. A customer has no influence and control above it. It is only directed by the system by random c

1.4 References

- Head First Design Patterns: A Brain-Friendly Guide (1st Edition) by Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson
- Head First Object-Oriented Analysis and Design (1st Edition) by Brett D. McLaughlin, Gary Pollice, Dave West

- Object-Oriented Analysis and Design with Applications (3rd Edition) by Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Boobi J. Young, Jim Conallen, Kelli A. Houston
- Effective Java 3rd Edition by Joshua Bloch

2. General Description

2.1 Product Perspective

The product should be similar to Monopoly Game. It requires to have some players and squares to simulate the game.

2.2 Product Functions

The product will simulate Monopoly Game by using the given information. After each step of the players, the program should display the current information of the players in the console. At the end of the game, the program must display the information of the winning player and the loser players on the console.

2.3 User Characteristics

The user should be able to change JSON file to give initial information to the program. Java should be installed in the user's computer. Also, the user should be able to run the game via terminal.

2.4 General Constraints

First constraint is simulation. Product can't be interrupted any process because it is meant to be run by the system itself. A second constraint is that the product cannot contain GUI.

2.5 Assumptions and Dependencies

The product can be run any operating system since it programmed in Java. During the development of JUnit and JSON Simple are used as dependencies. We added Spring and Logging as our libraries. We implemented Dependency Injection using Spring Core libraries. And we used logging to keep track of process of our simulation to detect errors in program.

3.2 Functional Requirements

3.2.1 Monopoly Game

3.2.1.1 Introduction

This is a classic monopoly game simulation.

3.2.1.2 Inputs

Required a JSon file to build game variables.

3.2.1.3 Processing

The data obtained from JSon file will process to build game.

3.2.1.4 Outputs

When a player’s turn comes, we print all the details.

3.2.1.5 Error Handling

There is no error handling.

3.3 Use Cases

3.3.1 Use Case #1

First, program reads instructions to start the game from a specific JSON File. After reading those infos, it prints them and starts the game. Then it initialises the players, giving them a unique piece and given balance.Then it starts the first round. In each turn it demands every Simulated Player(SP) to roll a dice. Before that it prints SP type, turn counter, cycle counter, SP location and current SP balance. After rolling two dice program controls the square that matches the total dice value. If faces of two dice are the same SP will have a chance of rolling one more time. This continues until SP does not rolls dice with same faces. Program moves the piece of current SP to the given square and perform whatever the specific square requires and prints SP location, SP balance, owner of the square, rent of the square and name of the square. Then it gives the turn to the other player this goes like this until the round is over. If a player has landed on a Jail square, he will be arrested and will not be able to make a new move until he rolls two dice with same face. A jail proposes three chance of rolling dice. After three roll jail will disseise some amount of money from SP who entered the jail. After the salvation of SP, he will have a chance of rolling dices without any waiting. When round is over program starts the round again. The only thing ceases this process is to detect that there is only one player remained. If an SP has no balance, it will be eliminated. However SP can rescue from this tricky situation by selling his properties until his balance is positive. If even after selling all his properties he is below zero then he will be eliminated from the game without doubt. When this occurs it prints the winner then the eliminated players.

3.4 Classes / Objects

3.4.1 Player

3.4.1.1 Attributes

```
static int totalRounds;
int balance, cycleCounter, totalDiceValue, roundValue,
currentDiceVal;
boolean bankruptFlag, isArrested;
BDice playerDice;
DPiece.PieceType pieceType;
ArrayList<BPropertySquare> propertySquares = new
ArrayList<>();
XYSeries playerDataset;
int roundCounter = 1;
private int[][] hasFullColor;
```

3.4.1.2 Functions

```
void checkAndUpdatePlayer(int currentDiceValue, Square
currentSquare)

boolean tryToSellProperty(BSquare currentSquare)

void sellSquare(BSquare square)

int rollDice()

boolean isPlayerCrossTheGoSquare()

boolean isPlayerBankrupted()

boolean buy(BPropertySquare currentSquare)

boolean isAbleToBuy(BPropertySquare currentSquare)

void sortSquares()

DPlayer getDPlayer()

void updateDataset(int turn, int money)

boolean controlHasFullColor(int color, int player)
```

3.4.2 Board

3.4.2.1 Attributes

```
static final int SQUARE_NUMBER = 40;

BSquare[] squares;

static BBoard boardInstance;

DInstruction instructionInstance;

ArrayList<BJailSquare> jailSquares;
```

3.4.2.2 Functions

```
void initSquares()

static Board getInstance()

ArrayList<BJailSquare> getJailSquares()
```

3.4.3 Square

3.4.3.1 Attributes

```
PropertyType pType;

String name="Default";

BPlayer ownerOfSquare = null;

int rent;

int price;
```

3.4.3.2 Functions

```
void performOnLand(Player player)

String getSQUARE_TYPE()

BPlayer getOwnerOfSquare()

void setOwnerOfSquare(BPlayer ownerOfSquare)
```

3.4.4 Dice

3.4.4.1 Attributes

Dice class includes no attributes.

3.4.4.2 Functions

```
int[] rollDice()

int[] rollDiceWithoutConstraint()

boolean checkIfDicesAreSame(int[] diceArray)
```

3.4.5 Observer

3.4.5.1 Attributes

Observer interface class includes no attributes.

3.4.5.2 Functions

```
void checkAndUpdatePlayer(int currentDiceValue, Square
currentSquare)

void listen()
```

3.4.6 Terminal

3.4.6.1 Attributes

```
CTerminal cTerminal
```

3.4.6.2 Functions

```
void printCurrentJSONFile()

void printBeforeRollDice(Player bPlayer)

void printDicesFaces(int[] diceValues)

void printLocationType(String squareLocation)

void printAfterRollDice(Player bPlayer)

void printLuckCard(String nameOfCard)

void printBuyProcess(BPlayer bPlayer, BSquare bSquare)

void printRentProcess(DPlayer dPlayer, BSquare bSquare)
void printWinnerPlayer(Player bPlayer)

void printGameOver(ArrayList<Player> eliminatedList)
```

3.4.7 Luck Cards

3.4.7.1 Attributes

```
private String CARD_INFO;
private int cardID;
```

3.4.7.2 Functions

```
public abstract void performForCard
```

3.4.8 Community Chest Cards

3.4.8.1 Attributes

```
String CARD_INFO;
int cardID;
```

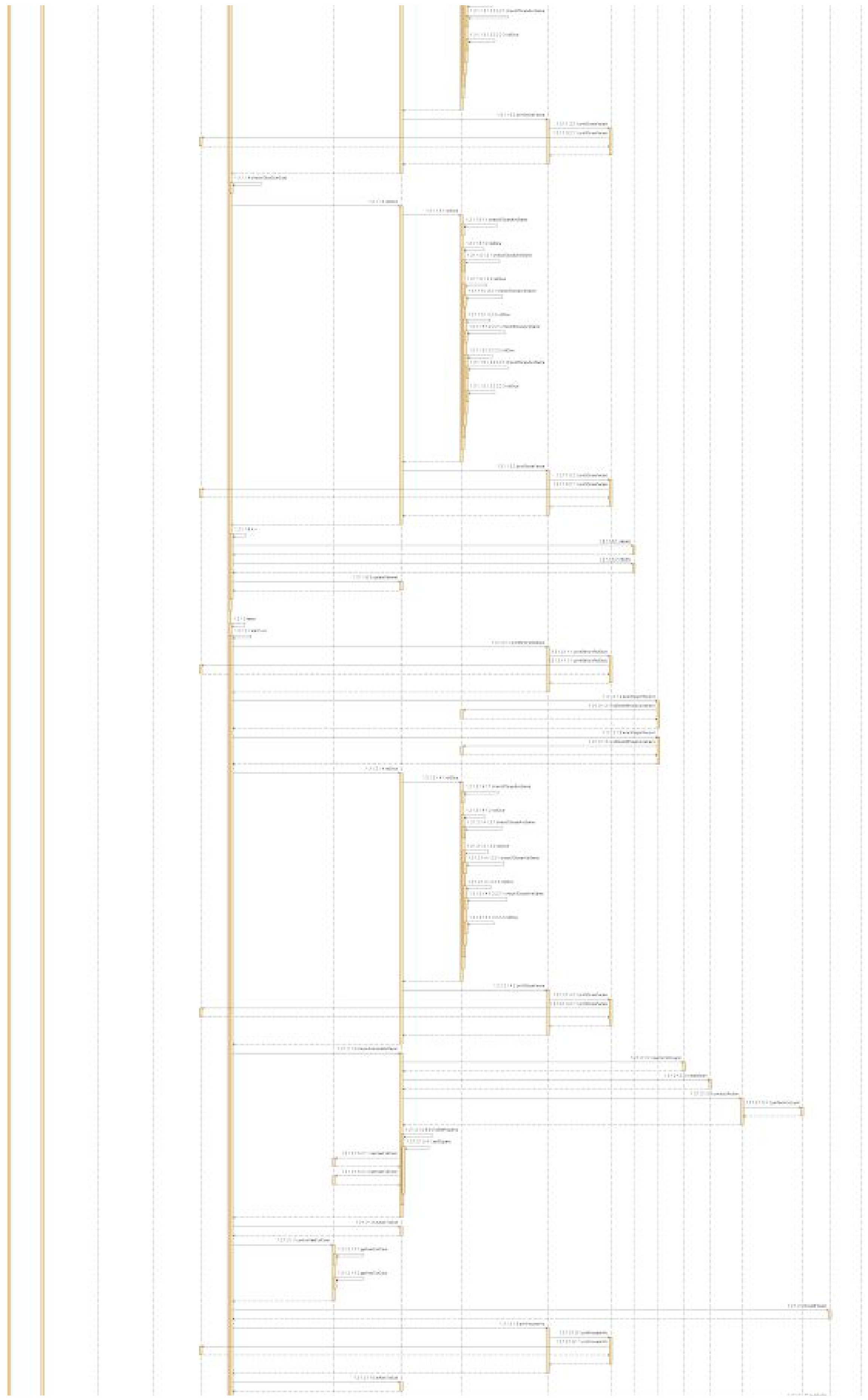
3.4.8.2 Functions

```
abstract void performForCard(DPlayer dPlayer);
```

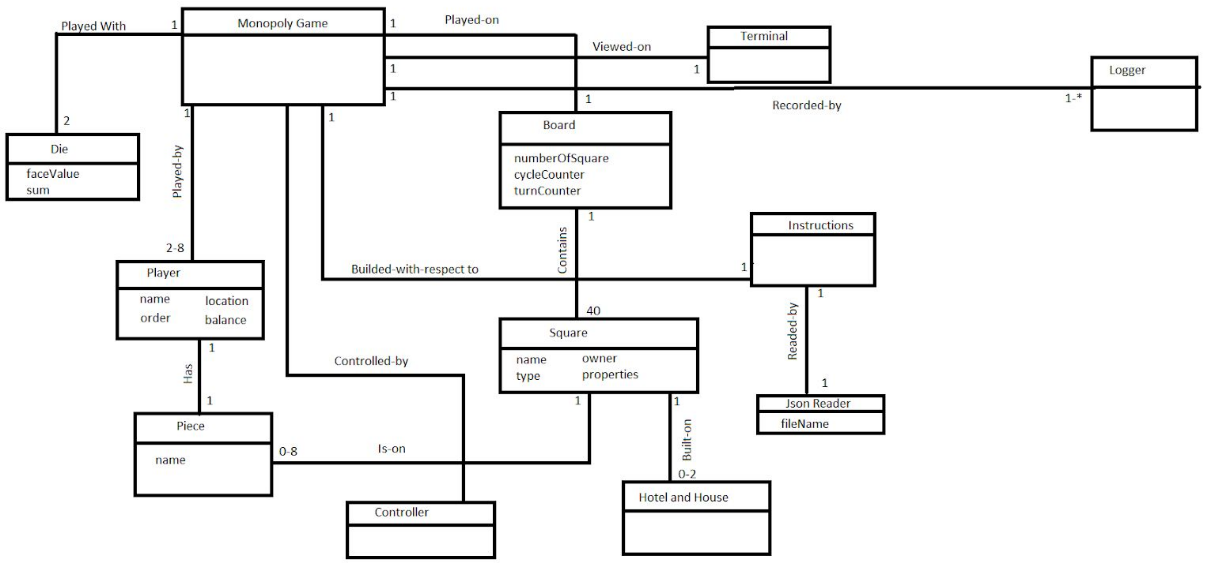
4. Analysis Models

You can access all models using the link: **Models in**

4.1 Sequence Diagram

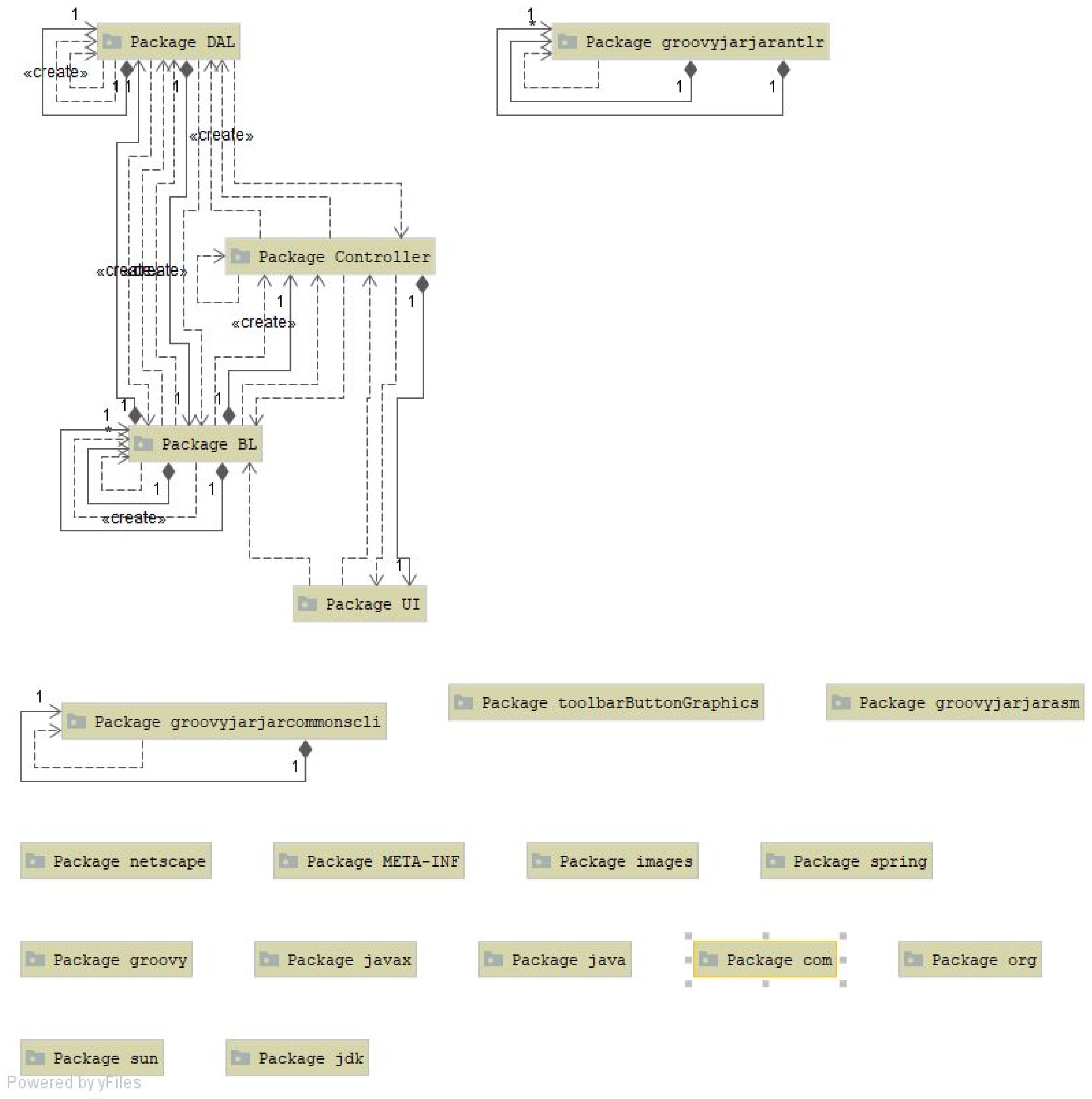


4.2 Analysis Domain Model

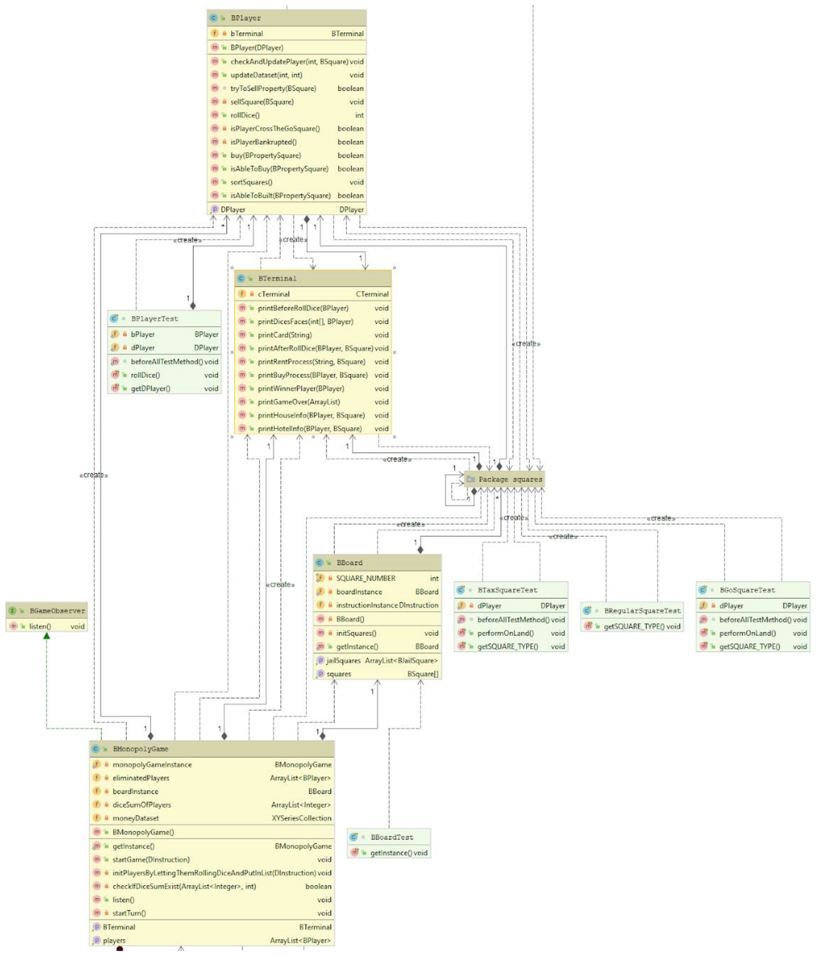


4.3 UML Diagrams

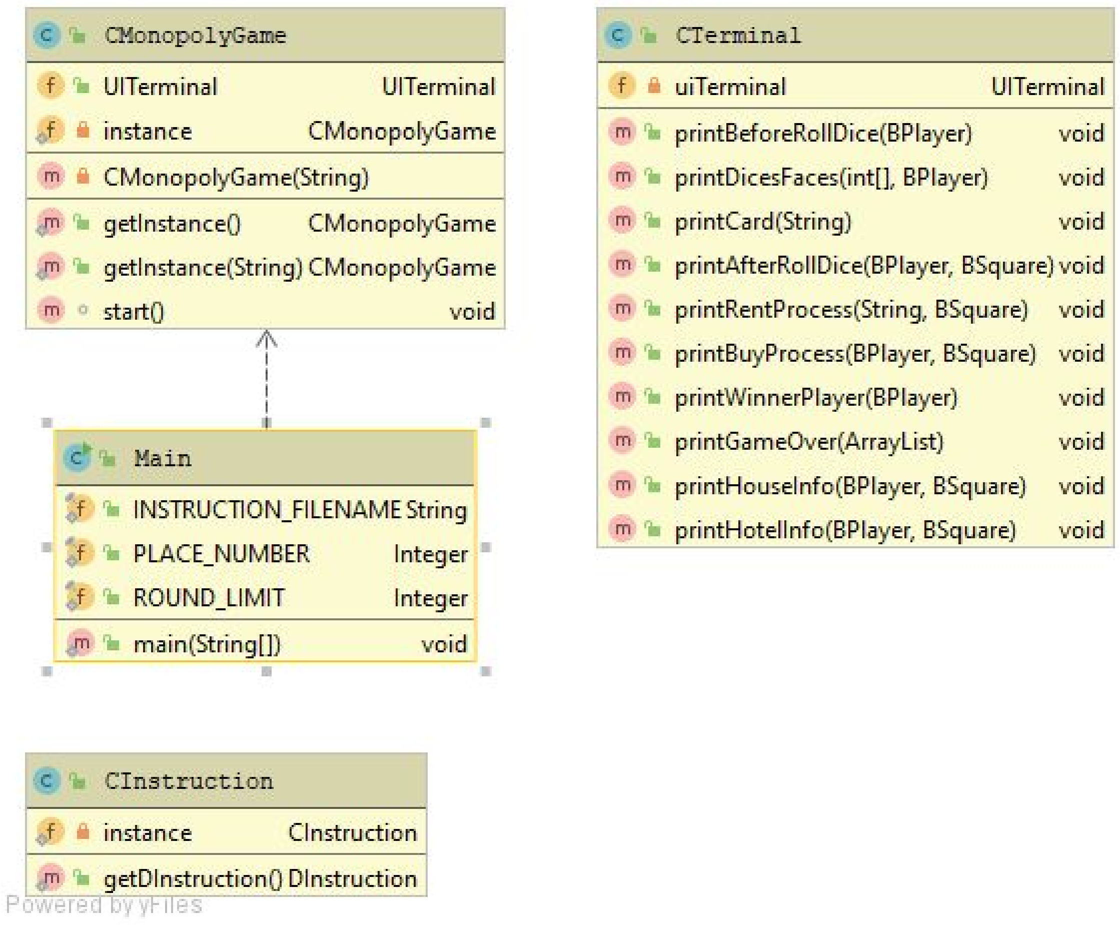
Package Dependencies



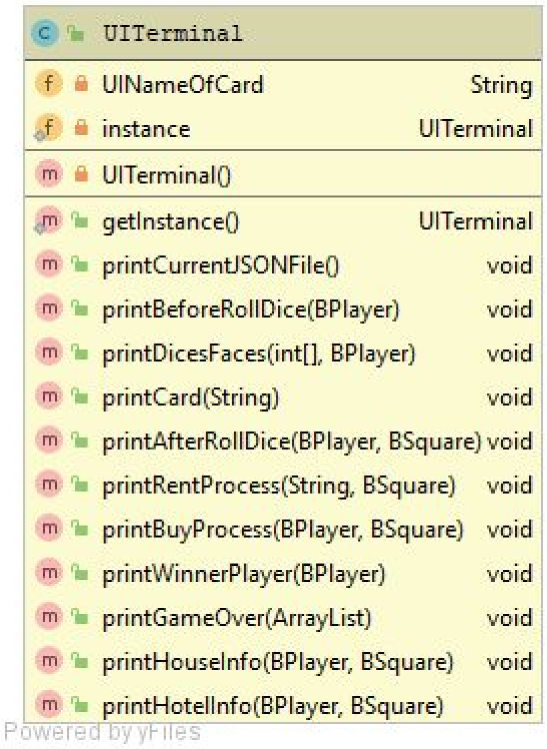
BL package UML Diagram


BPlayer
bTerminal BTerminal
BPlayer(DPlayer)
checkAndUpdatePlayer(int, BSquare) void
updateDataset(int, int) void
tryToSellProperty(BSquare) boolean
sellSquare(BSquare) void
rollDice() int
isPlayerCrossTheGoSquare() boolean
isPlayerBankrupted() boolean
buy(BPropertySquare) boolean
isAbleToBuy(BPropertySquare) boolean
sortSquares() void
isAbleToBuilt(BPropertySquare) boolean
DPlayer DPlayer
BPlayerTest
bPlayer BPlayer
dPlayer DPlayer
beforeAllTestMethod() void
rollDice() void
getDPlayer() void
BTerminal
cTerminal CTerminal
printBeforeRollDice(BPlayer) void
printDicesFaces(int[], BPlayer) void
printCard(String) void
printAfterRollDice(BPlayer, BSquare) void
printRentProcess(String, BSquare) void
printBuyProcess(BPlayer, BSquare) void
printWinnerPlayer(BPlayer) void
printGameOver(ArrayList) void
printHouseInfo(BPlayer, BSquare) void
printHotelInfo(BPlayer, BSquare) void
Package squares
«create»
1
BGameObserver
listen() void
BBoard
SQUARE_NUMBER int
boardInstance BBoard
instructionInstance DInstruction
BBoard()
initSquares() void
getInstance() BBoard
jailSquares ArrayList<BJailSquare>
squares BSquare[]
BTaxSquareTest
dPlayer DPlayer
beforeAllTestMethod() void
performOnLand() void
getSQUARE_TYPE() void
BRegularSquareTest
getSQUARE_TYPE() void
BGoSquareTest
dPlayer DPlayer
beforeAllTestMethod() void
performOnLand() void
getSQUARE_TYPE() void
BMonopolyGame
monopolyGameInstance BMonopolyGame
eliminatedPlayers ArrayList<BPlayer>
boardInstance BBoard
diceSumOfPlayers ArrayList<Integer>
moneyDataset XYSeriesCollection
BMonopolyGame()
getInstance() BMonopolyGame
startGame(DInstruction) void
initPlayersByLettingThemRollingDiceAndPutInList(DInstruction) void
checkIfDiceSumExist(ArrayList<Integer>, int) boolean
listen() void
startTurn() void
BTerminal BTerminal
players ArrayList<BPlayer>
BBoardTest
getInstance() void

Controller package UML Diagram

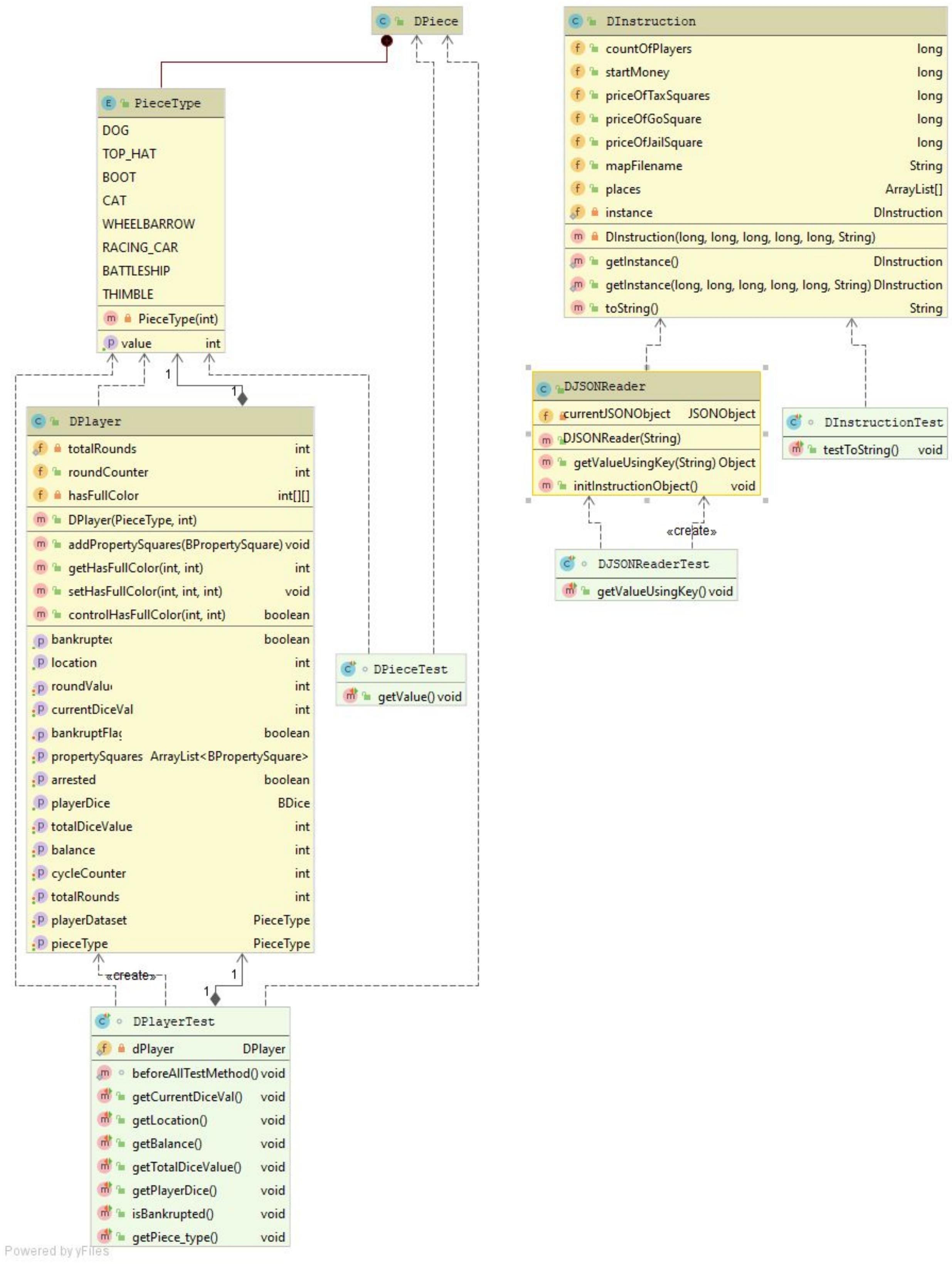

CMonopolyGame
UITerminal UITerminal
instance CMonopolyGame
CMonopolyGame(String)
getInstance() CMonopolyGame
getInstance(String) CMonopolyGame
start() void
Main
INSTRUCTION_FILENAME String
PLACE_NUMBER Integer
ROUND_LIMIT Integer
main(String[]) void
CInstruction
instance CInstruction
getDInstruction() DInstruction
Powered by yFiles
CTerminal
uiTerminal UITerminal
printBeforeRollDice(BPlayer) void
printDicesFaces(int[], BPlayer) void
printCard(String) void
printAfterRollDice(BPlayer, BSquare) void
printRentProcess(String, BSquare) void
printBuyProcess(BPlayer, BSquare) void
printWinnerPlayer(BPlayer) void
printGameOver(ArrayList) void
printHouseInfo(BPlayer, BSquare) void
printHotelInfo(BPlayer, BSquare) void

UI package UML Diagram



UITerminal	
UINameOfCard	String
instance	UITerminal
UITerminal()	
getInstance()	UITerminal
printCurrentJSONFile()	void
printBeforeRollDice(BPlayer)	void
printDicesFaces(int[], BPlayer)	void
printCard(String)	void
printAfterRollDice(BPlayer, BSquare)	void
printRentProcess(String, BSquare)	void
printBuyProcess(BPlayer, BSquare)	void
printWinnerPlayer(BPlayer)	void
printGameOver(ArrayList)	void
printHouseInfo(BPlayer, BSquare)	void
printHotelInfo(BPlayer, BSquare)	void

DAL package UML package



Stakeholders

- Murat Can GANİZ (Customer)
- Serap KORKMAZ (Customer)
- Ahmet LEKESİZ (Programmer)
- Mücahit TANACIOĞLU (Programmer)
- Muhammed Bera KOÇ (Programmer)
- Salih ÖZYURT (Programmer)