Danmarks Tekniske Universitet

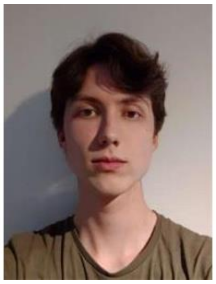
CDIO 3

(62531) Development Methods for IT-systems,
(62532) Version Control and Test Methods,
(02314) Introduction to Programming

Balder Jacobsen, s235094

Setare Izadi, s232629

Viktor Steffensen, s214707



Tobias N. Frederiksen, s235086

24. november 2023

1 Summary

In this project, we were tasked by IOOuterractive, with creating a Monopoly Jr. game coded in Java. This report details our preparations for the project and outlines our development methods. Our primary goal was to replicate the core mechanics of the Monopoly Jr. game in an object-oriented programming language while introducing and enhancing interesting features. We adopted the GRASP programming pattern throughout development and rigorously tested our functions to ensure that the product met our high standards and those of the client. Notably, our implementation introduces an automation of the banking system, this is done to make the game easier and more approachable.

2 Hourly accounting

Each member of the group has worked approximately 2 hours after each lecture tuesdays and fridays, from october 31st to november 24th. Each member has also worked an hour weekly at home, meaning each member has worked approximately 20 hours total.

3 Table of contents

4 Introduction

In our new CDIO 3 project, we've recieved yet another request by IOOuterActive. And this time their vision is for us to develop a Monopoly Junior game.

In the game, Chance Cards introduce unpredictability, guiding player actions with drawn instructions, influencing movements, and property acquisitions. The circular game board ensures seamless player movement with each dice roll, offering diverse properties (boardfields) like Burgerbaren, Pizzariaet, and Slikbutikken, presenting unique challenges and opportunities. Designed for 2 to 4 players, the objective is strategic navigation, property acquisition, and wealth accumulation, adding a competitive edge to the gameplay.

The game's architecture is visualized through design class diagrams, class diagrams, and domain models, providing a clear blueprint for development.

Sequence diagrams map out the flow of interactions, ensuring the game's logic aligns with the envisioned gameplay. Robust testing is carried out through JUnit -and user testcases, validating the functionality of different game components.

As we navigate this development journey, the objective is to create an immersive Monopoly Junior game, where strategic gameplay and diverse property types meet. By combining Java programming with UML, IOOuterActive strives to deliver a polished and engaging gaming experience for players.

5 Requirement analysis

We began our requirement analysis by finding and clarifying all the requirements we got from the costumer.

- The game is a dicegame.
- The game is played by 2-4 players.
- The die is 6-sided and fair.
- Each player starts with a certain amount of money based on the number of players.
- The game ends if a player is unable to purchase a field or pay rent or chance fees.
- If two or more players have an equal ammount of money, then value of their field is also counted.
- Each of the chance cards should do the specific action.
- Collect 2M at start.
- The sum of the diceroll determines the players movement.

Demands for versioncontrol:

- IOOuteractive wants to be able to see who did what - git-repo.
- Documentation for the platforms parts with a versionnumber, so it can be recreated and used later.
    - Commit often - atleast everytime a small portion of code is made or enhanced.
    - Every commit needs a small description of what has been made/changed.
    - Each commit has to only contain relevant changes. (If you fix a bug AND a spelling mistake - make two seperate commits with comments in both.)
    - Only code that works is to be commited into main.
- IOOuteractive would also like to know the system requerments (Does anything need to be installed?).

Demands for specific tests, methods and diagrams:

- Use case diagram.
- Use case descriptions (fully flexed).
- Domain model.
- Systsem sequence diagram (only the most important operations.)
- Sequence diagram.
- Design class diagram.
- Minimum one usertest. The user shouldn't be able to code.
- Minimum three testcases with testprocedure and rapport.
- Minimum one Junit test. Including code coverage documentation.
- Other relevant tests.

Configuration:

IOOuteractive wants to know what demands there are to installed programs and OS. They need a description of minimumrequirements and a tutorial in how the sourcecode is compiled, installed and removed. This includes a description of how the code is imported from a git-repo.

6 Project planning

After we had our all of our requirements set, we moved on to our project planning phase.

We have chosen to start by identifying, the actors in this project.

6.1 Actors

We have identified the actors in the program as:

- Player_1
- Player_2
- Player_3
- Player_4
- Bank (For keeping track of money)
- PC (Running)
- Keyboard (Input)
- Monitor (Visual output)

6.2 Usecases

From here we wrote a usecase

Use case:

Scope	Monopoly JR
Level	User goal
Primary actor	Players
Stakeholders and Interests	- Players: Want to play the game.
Preconditions	Players must have a computer capable of running the program.
Succes Guarantee	The game is played to the end with no crashes.
Main Succes Scenario	1. The game is launched. 2. The players picks their piece's / colors. 3. The Bank distributes money to each player. 4. Player one is determined 5. Player one starts the game, by rolling the dice. 6. Player moves the ammout of eyes on the dice. 7. Field effect is determined, and executed depending on the type of field. 8. Player turn over, next player turn start. 9. Player turns keep going until one player cant pay for or buy anything. 10. If a player lands on a chance field, execute the effect on the chance card, if this card is a get out of jail free, save this to the player. 11. When a player crosses start, they recive 2M. 12. If a player owns two fields of the same color the value of these fields are double. 13. If player goes to jail, they have to pay 1M to get out unless they have a get out of jail chancecard. 14. If player doesn't have any money to buy field or pay rent the game ends and the player with most money wins
Frequency of Occurance	

7 Design

7.1 Domain model

Our domain model illustrates the most important concepts and their attributes, and the relationships between the concepts.

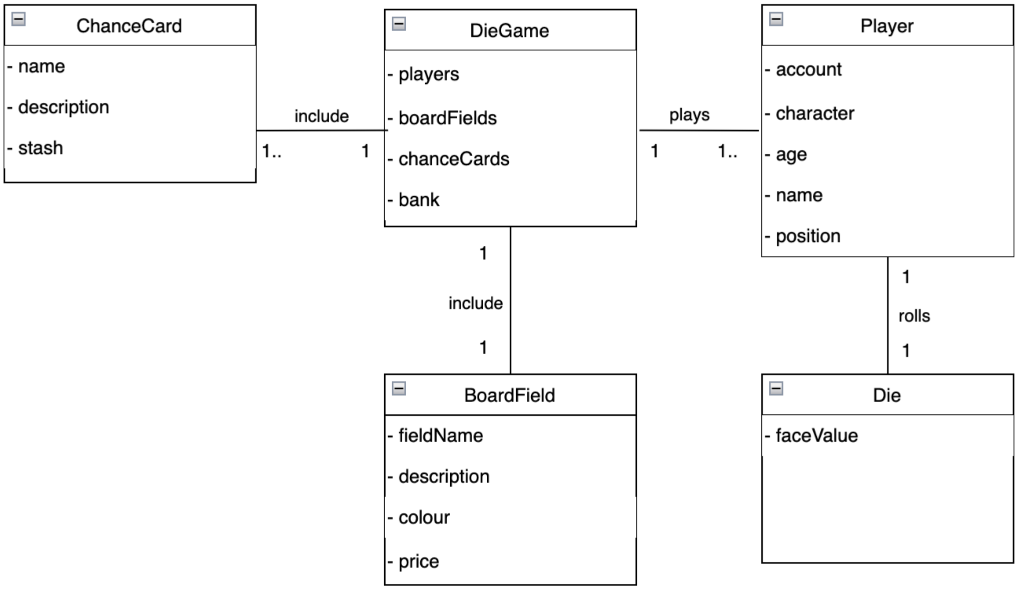


Figure 1: Domain model

7.2 Class diagram and design class diagram

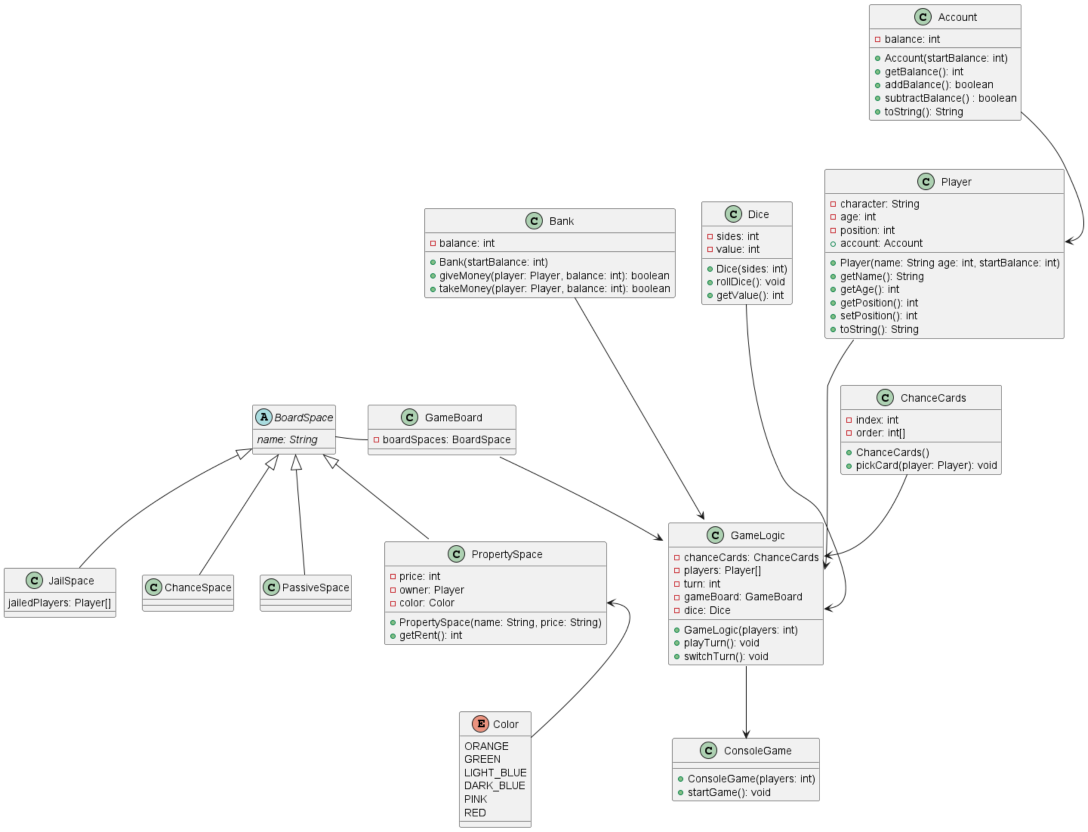


Figure 2: Class diagram

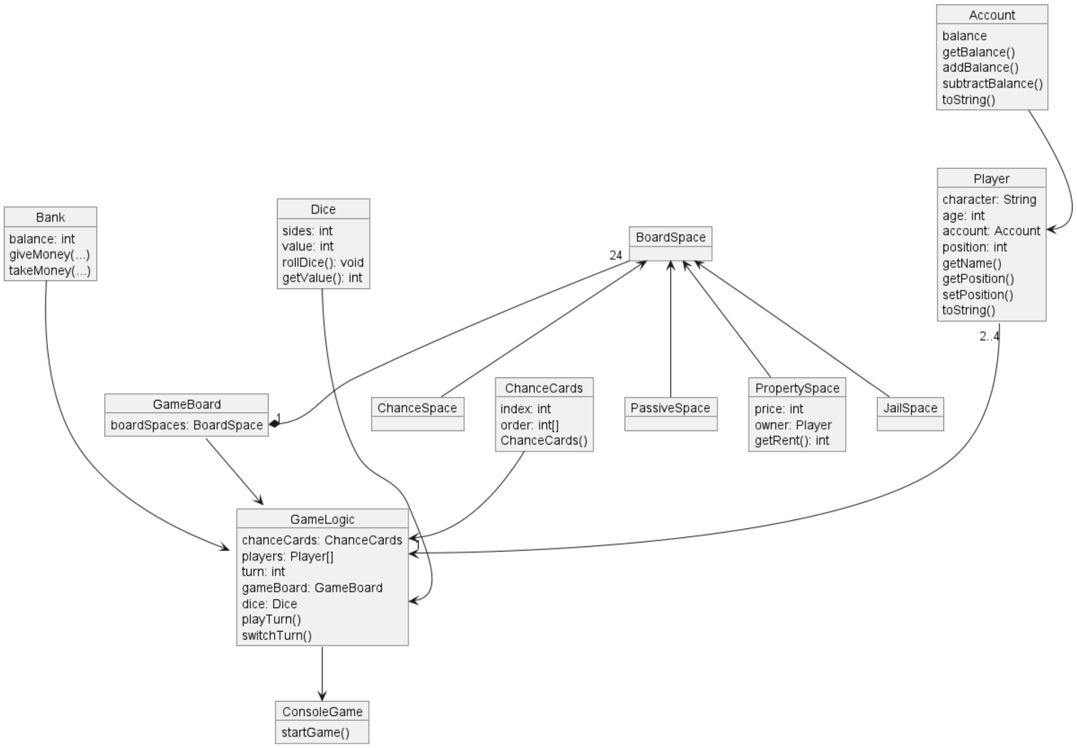


Figure 3: Design class diagram

Our class diagram and design class diagram shows the relationship between our major classes, their atrributes and method signatures. They help give an overview of the structure of the program.[1]

[1] For potential better quality inspect the prictures of the diagrams directly in the folders

7.3 Use case diagram

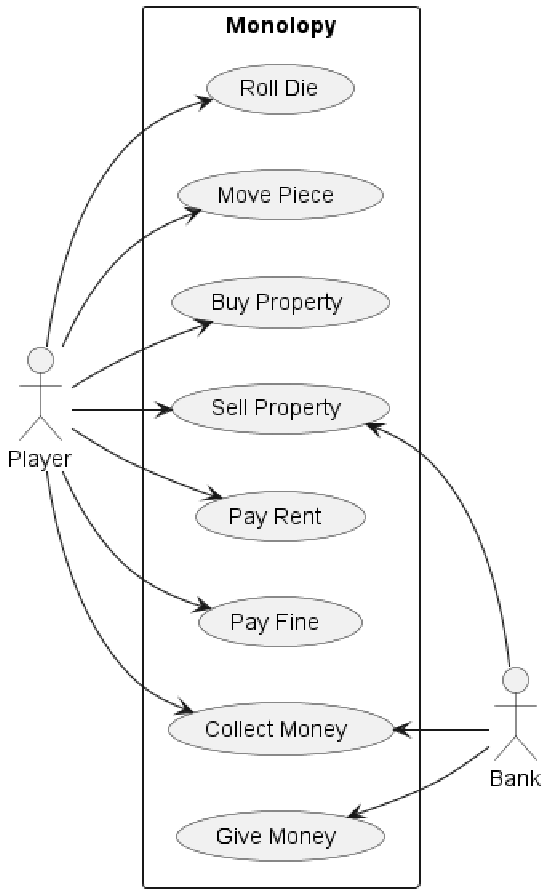


Figure 4: Use case diagram

Our use case diagram shows the use cases for the actors, wihch are players. Normally in monolopy one player would also assume the role of a bank, however we have chosen to let the system be the bank, as the payments are automated in our program.

7.4 Sequence diagram

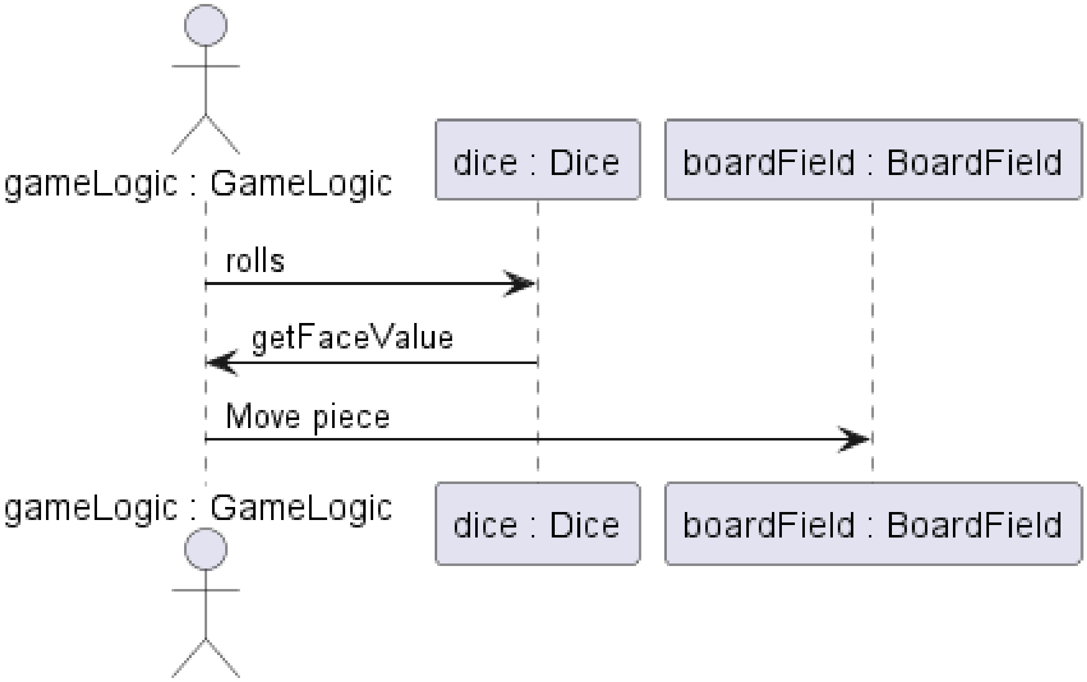


Figure 5: Sequence diagram

This sequence diagram shows a simple action. A player tells the game, they would like to roll the die. The GameLogic class then rolls the die, wihch then returns a value that determines the amount of spaces/fields the players gamepiece moves on the boardfield.

7.5 System sequence diagram

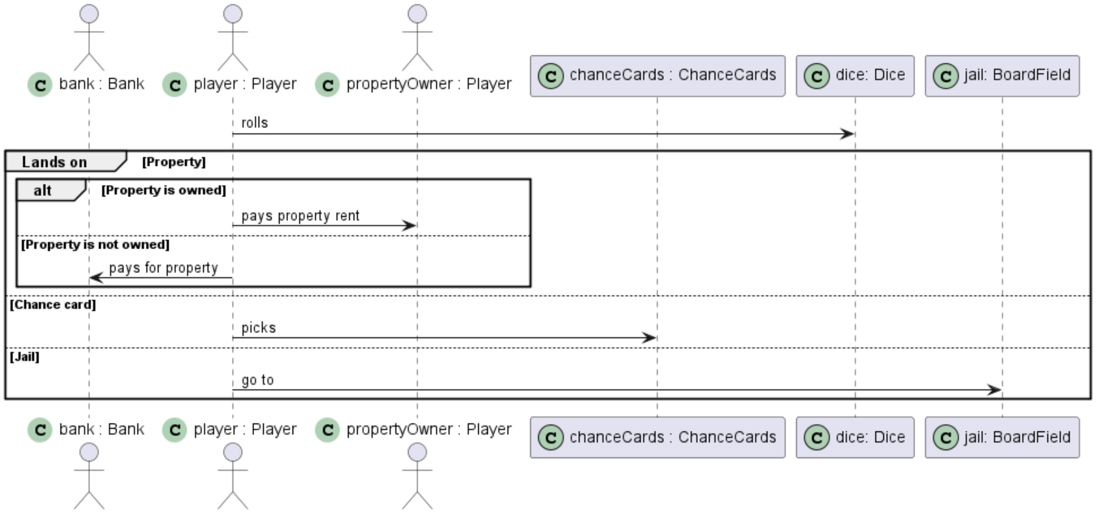


Figure 6: System sequence diagram

The SSD shows shows what happens based om the type of field a player lands on. First the player rolls the die. Then based on which field they land on different things can happen. If they land on owned property, they pay rent. If they land on unowned property, they buy it. There are also other types of fields, where we have chosen to show chancecards and the ”Go to jail” fields.

8 Implementation

As per IOOuteractive's request we have been working on the code using, the GRASP patterns. This helped us use functions from differet classes, to simplify and organise all of our code. We have chosen to make the bank a part of the system, so a player doesn't have to assume the role of the bank. Furthermore we have excluded the characters, such as boat, dog, cat and car, since we didn't find it nescessary, as the game is run in a terminal, so moveable characters are redundant. As a consequence of this, we instead choose to remove the four chance cards, which directly impacts each character e.g.:

> "Give this card to the car, and draw another chance card.
> Car: On your next turn, drive to any unoccupied field and buy it. If there aren't any unoccupied fields, buy one from another player."

At the end of the project we looked into creating a GUI and mannaged to setup a minor frame work for this. Although the development went smoothly, we choose to put this functionality on ice for the time being and foucs our energy to polish the rest of our project to be as good as possible.

9 Documentation

9.1 Configuration control

Development platform
In our development platform we've used the following software:
- Apple macOS Ventura
- Windows 11
- Visual Studio Code (VSC)
- Java (Version "20.0.2" and "17.0.8")
- (Maven)

Production platform
The production platform is all the software that's used to run our final program:
- Operating system that is able to run Java
- Java Version "20.0.2" and "17.0.8")

If you want to develop the program further, It's necessary to use other configurations:
- Apple macOS Ventura
- Windows 11
- Visual Studio Code (VSC). It's possible to use other integrated development environments (IDE's), yet there's settings in VSC which provides the best advantage to write the code.

10 Testing

10.1 Usertest

10.2 Testcases

With all of our requirements found we began to design our testcases. We chose to design these tests towards the most importants functions of our product.

Test-cases: Testcase 1

Testcase ID	TC01
Summary	Test that the Bank distributes the right amount of money to each player during the start of the game.
Requirements	Requirement Specifiaction RS04 and RS02.
PreConditions	Two players have been added to the game.
PostConditions	The game is still going.
Test Procedure	1. The players pick their game pices 2. The players input the requied info. 3. The game is started.
Test data	Player one (Name and age), Player two (Name and age)
Expected Result	The number of players is 2 which means the start balance is 20 M
Actual result	The number of players is 2 which means the start balance is 20 M
Status	Completed
Tested by	Setare izadi
Date	21/11-23
Test enviroment	Visual Studio Code: Version: 1.84.2 (user setup) Commit: 1a5daa3a0231a0fbba4f14db7ec463cf99d7768e Date: 2023-11-09T10:51:52.184Z Electron: 25.9.2 ElectronBuildId: 24603566 Chromium: 114.0.5735.289 Node.js: 18.15.0 V8: 11.4.183.29-electron.0 OS: WindowsN T x6410.0.22621.Windows11pro, Version 0.1.1, Built 11-21-2023 14:54

Testcase 2

Testcase ID	TC02
Summary	Test that the player chance cards work.
Requirements	A player lands on chance and ”pulls” the card
PreConditions	A player lands on chance and ”pulls” the card
PostConditions	The game continues
Test Procedure	1. The card is pulled 2. The required player gets the card 3. The Player pulls a new card and continues thier turn. 4. The Player who got the card follows the directions of the card.
Test data	Chance cards and board fields
Expected Result	The player goes to an empty field and buys it, if theres is no empty fields they pick one owned field and buys it from the player.
Actual result	
Status	
Tested by	
Date	
Test enviroment	

This is the test case for the unsuccessful test.

Testcase 3

Testcase ID	TC03
Summary	Test that the game will end if a player cant pay the price of the field they land on.
Requirements	
PreConditions	The game has been going and a one player is low on money
PostConditions	The game is over and the winner have been found
Test Procedure	1. A player is low on money. 2. The player's turn starts. 3. The player moves. 4. The player cant pay the price of the field. 5. The game ends.
Test data	Player accounts
Expected Result	The game ends and the positions is decied buy the value of their account.
Actual result	The game continues and the ”losing” Player recives more money.
Status	Not Complete
Tested by	Balder and Viktor
Date	21/11-2023
Test enviroment	Visual Studio Code: Version: 1.84.2 (user setup) Commit: 1a5daa3a0231a0fbba4f14db7ec463cf99d7768e Date: 2023-11-09T10:51:52.184Z Electron: 25.9.2 ElectronBuildId: 24603566 Chromium: 114.0.5735.289 Node.js: 18.15.0 V8: 11.4.183.29-electron.0 OS: Windows$_NTx6410.0.22621.Windows11pro$, Version 0.1.1, Built 11-21-2023 14:54

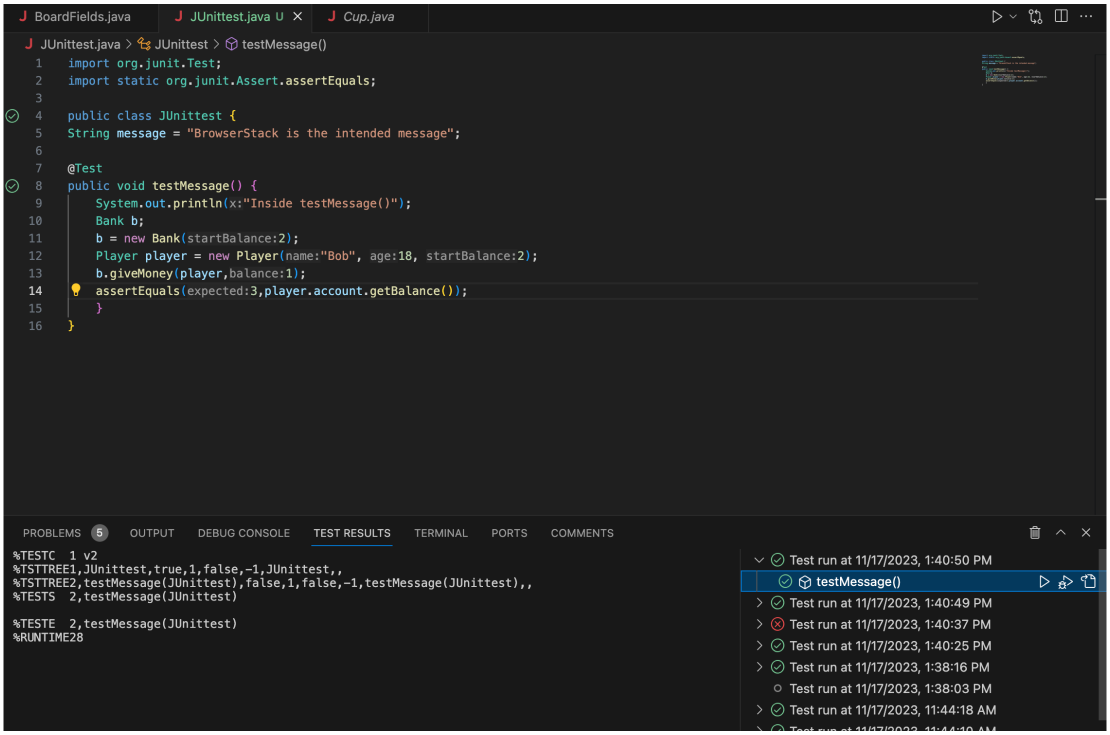
This is the test case for the succesful test.

Testcase 3.5

Testcase ID	TC3.5
Summary	Test that the game will end if a player cant pay the price of the field they land on.
Requirements	
PreConditions	The game has been going and a one player is low on money
PostConditions	The game is over and the winner have been found
Test Procedure	1. A player is low on money. 2. The player's turn starts. 3. The player moves. 4. The player cant pay the price of the field. 5. The game ends.
Test data	Player accounts
Expected Result	The game ends and the positions is decied buy the value of their account.
Actual result	The game ends and the winner is displayed.
Status	Approved
Tested by	Balder and Viktor
Date	11/21-2023
Test enviroment	Visual Studio Code: Version: 1.84.2 (user setup) Commit: 1a5daa3a0231a0fbba4f14db7ec463cf99d7768e Date: 2023-11-09T10:51:52.184Z Electron: 25.9.2 ElectronBuildId: 24603566 Chromium: 114.0.5735.289 Node.js: 18.15.0 V8: 11.4.183.29-electron.0 OS: Windows$_NTx6410.0.22621.Windows11pro$, Version 0.1.1, Built 11-21-2023 - 15:23

10.3 JUnit test

We made a unt test ensure that a balance is updated correctly when money is given. The java file has also been uploaded to our Git repository.



```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class JUnittest {
String message = "BrowserStack is the intended message";

@Test
public void testMessage() {
    System.out.println(x:"Inside testMessage()");
    Bank b;
    b = new Bank(startBalance:2);
    Player player = new Player(name:"Bob", age:18, startBalance:2);
    b.giveMoney(player,balance:1);
    assertEquals(expected:3,player.account.getBalance());
    }
}
```

```
%TESTC  1 v2
%TSTTREE1,JUnittest,true,1,false,-1,JUnittest,,
%TSTTREE2,testMessage(JUnittest),false,1,false,-1,testMessage(JUnittest),,
%TESTS  2,testMessage(JUnittest)

%TESTE  2,testMessage(JUnittest)
%RUNTIME28
```

```
✓ Test run at 11/17/2023, 1:40:50 PM
    ✓ testMessage()
✓ Test run at 11/17/2023, 1:40:49 PM
✗ Test run at 11/17/2023, 1:40:37 PM
✓ Test run at 11/17/2023, 1:40:25 PM
✓ Test run at 11/17/2023, 1:38:16 PM
○ Test run at 11/17/2023, 1:38:03 PM
✓ Test run at 11/17/2023, 11:44:18 AM
```

Figure 7: JUnit test

11 Conclusion

In conclusion, our journey in developing the Monopoly Junior game for IOOuterActive has been marked by thorough planning, innovative design, and careful testing.

With our robust testing, including JUnit and user test cases, we've guaranteed the functionality and reliability of the diverse game components, like the properties that can be bought throughout the boardfields. The boardfields has created a compelling and unpredictable gaming experience for 2 to 4 players, and this aspect has been carefully programmed to ensure a circular gaming flow.

As a result of our various UML diagrams, we've also effectively crafted a blueprint for our Monopoly game, and this blueprint has now materialized into a real, functioning and immersive Monopoly Junior game.

As we finish up, we're confident that what we've built will go beyond what IOOuterActive asked for, giving players a game that's not just good but great.

12 Appendix

12.1 Literature