Danmarks Tekniske Universitet

DTU

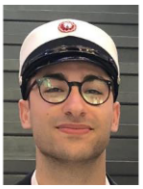
02314 | 62531 | 62532

Introductory Programming | Development methods for IT-systems | Version control and test methods

CDIO 3
Group 7

Friday 24th November, 2023 23:59

Authors


Mustafa Baker - s235092


Mads Hartwich - s235102


Adel Wisam - s236076


Emil Savino - s235096


Elias Larsen - s235125


Frederik Bastrup - s235117

1 Hourly Accounting

The group has both spent time working together and separately. Some of us have spent more time on some code than others and some have spent more time on modelling and the report. These roles have also changed from time to time, so generally everybody has had a part in each segment of the project. We have all spend almost the same amount of time on the project. We have spent around 8 hours a week per person during this project.

Contents

2 Summary

The group has been using Java language, Git and GitHub to develop the 2-4 player Monopoly Junior game. In the code there have been taken use of OOAD (Object-Orientated programming). This report includes everything from the project. This means all of the thoughts the group has had before programming, UML diagrams, models and methods, programming of the code it-self, testing and all the thoughts after the project. It also includes how the group has done the work during the project and the GitHub repository for the project in order to see our changes to the code and diagrams. We have also chosen parts of the code and described those specific parts.

3 Introduction

The group has been making a Java project, which is a Monopoly Junior game as we know it from the classic board game, but as a software program with simpler rules. The group has not used all the same rules as in the standard Monopoly junior game, but has chosen to remove some of them and only keep the most important ones. The game can be played by 2 - 4 players. Everyone is against each other. The starting balance depends on how many players are playing the game. One player rolls two die and moves the amount of fields the die show. When landing on a field, depending on the type, you buy the field you land on if you have money enough to do so. Otherwise you can't. Then the turn is passed on and it is the next players turn to roll the two die. If someone lands on an owned field, the player who lands on it will have to pay rent to the player who owns it. The game continues until a player is bankrupt. The money of the remaining players is then added up and the one with the most money wins the game.

4 Version control

The group has created and used a GitHub repository[3] for the project. All the members of the group have access, which means every member of the group can see the files in the repository, edit the files, add new files and delete files. They can also see who has committed what and when. The group have used the development branch to have all the code, which is under development and doesn't necessarily work yet. The development branch is then merged to the main branch when the code is working as intended. In the repository there is also a README-file, which contains information about the game, how it works and all the rules for the game. There is also a short description on how to start and play the game. The repository also contains all of the UML diagrams used in the report, changes to these can also be seen in the commit history on out GitHub repository.

5 Requirement/Analysis

There was given some requirements for the software program, but these requirement were quite loose, meaning that we can change and modify rules as we like. Therefore we have changed some of the rules, that we as a group, did not feel matched the software version of Monopoly Jr. For instance, we have removed the player tokens as we felt it didn't make sense for this version of monopoly junior, as we don't use a GUI. Additionally we have removed the chance cards which was supposed to be giving to another player, as our limited resources didn't allow it to be implement it. The rules we have decided to implement is described in the next segment.

Requirement specification

- The game has to include the game rules in English
- The game has to include a gameboard.
- The game has to include chance cards.
- The player will stay on the field they land on until their next turn and then move from that field next time the roll the dices (this is unless a chance card or prison field states something else).
- The game has to be played between 2 - 4 players.
- The fields need to have some kind of action when the player lands on it, which effects the status of the player and/or the game.
- The player has to buy a field, when they land on it (if it's not already owned by another player or they do not have the balance to support the purchase).
- The player with the most money wins, when a player goes bankrupt.
- The player has to be able to both receive and lose money.
- The player has to be able to get in - and out of jail.
- The player has to be able to move their character the amount of the fields the die shows.

Use cases

- Roll die
- Buy tile
- Pay owner
- Go to prison
- Go bankrupt
- Receive money
- Lose money
- Get chance card
- Move character
- Lose game

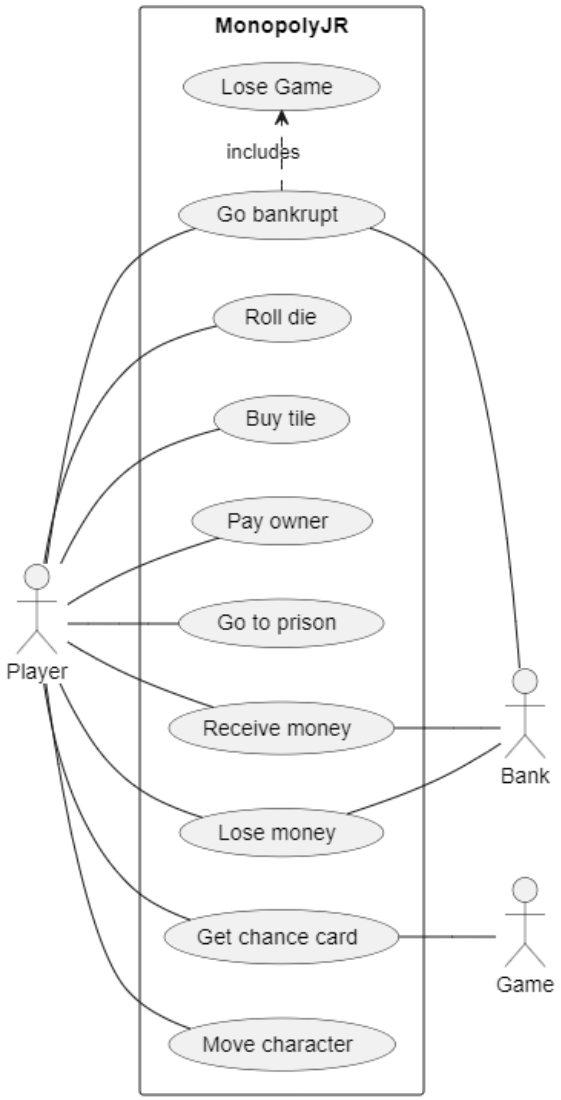
Fully dressed use case The group have made a fully dressed use case. This is one of the use cases for the Monopoly Junior game, they wanted to do as a fully dressed use case and get more into details with that specific use case. The group have chosen to do use case 2 "buy tile" as a fully dressed.

Use case	Description
Use case name	Use case 2 - Buy tile
Scope	Monopoly JR game
Level	User goal
Primary actor	Player
Stakeholders and interest	The customer (IOOuterActive), their interest is to create a monopoly jr game, that can be played from the console.
Preconditions	The game must be running, and it must be the players turn. Player also has to land on a purchasable tile.
Success guarantee	The player lands on a buyable tile, and has enough money to buy said tile.
Main success scenario	The user opens the game, and starts running it. When it gets to the user's turn, they roll a dice and land on a tile that hasn't been bought yet. The user has enough money to buy the tile, and proceeds to do so. Turn is then passed on or carried on depending on the initial roll
Extensions	2A. Player lands on a buyable tile, and doesn't have enough money. 2B. Player doesn't land on a buyable tile 2C. Player is in prison. 2D. Player is bankrupt 2E. Players turn is not passed on correctly 2F. Player buys a tile, but nothing happens.
Special requirements	Player must read the README file to know how to operate the system and the player must read the monopoly jr rules.

5.1 Diagrams/Models

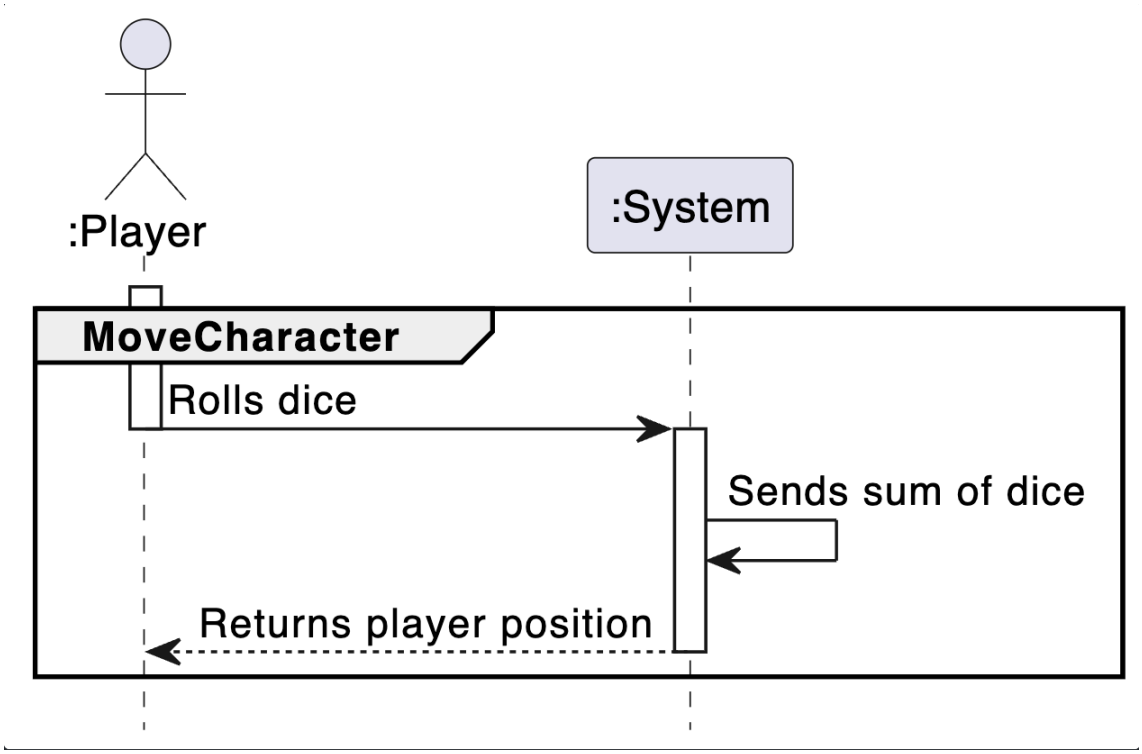
Use case diagram

The use case diagram is one of the models, that the group have chosen to use for the project. It's used to show what use cases the actors have to go through and who has to go through what. The primary actor is the player and the secondary actors is the bank and game.



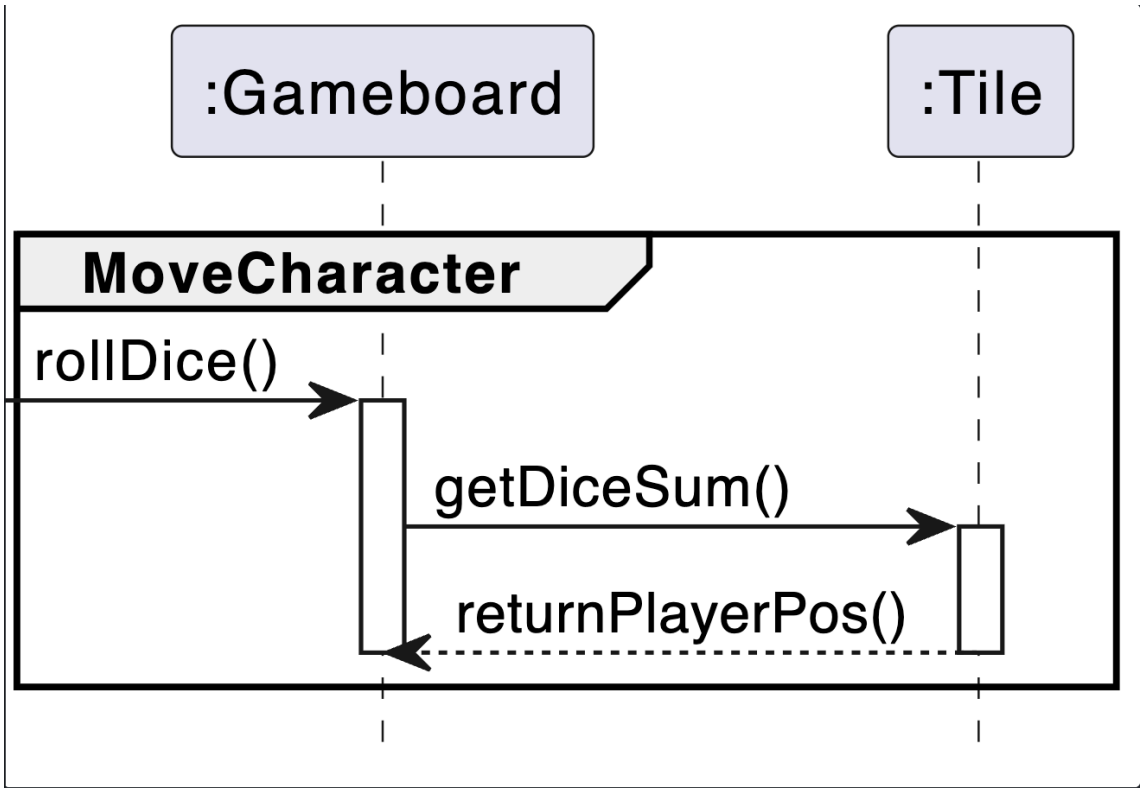
System Sequence diagram

The group has also used the system sequence diagram for the project. this diagram is used to show how the primary, which is the user, interacts with the program. This means how the user walks through the program from the start to the end. So the diagram shows the different parts of the Monopoly Junior game the player have to interact with.



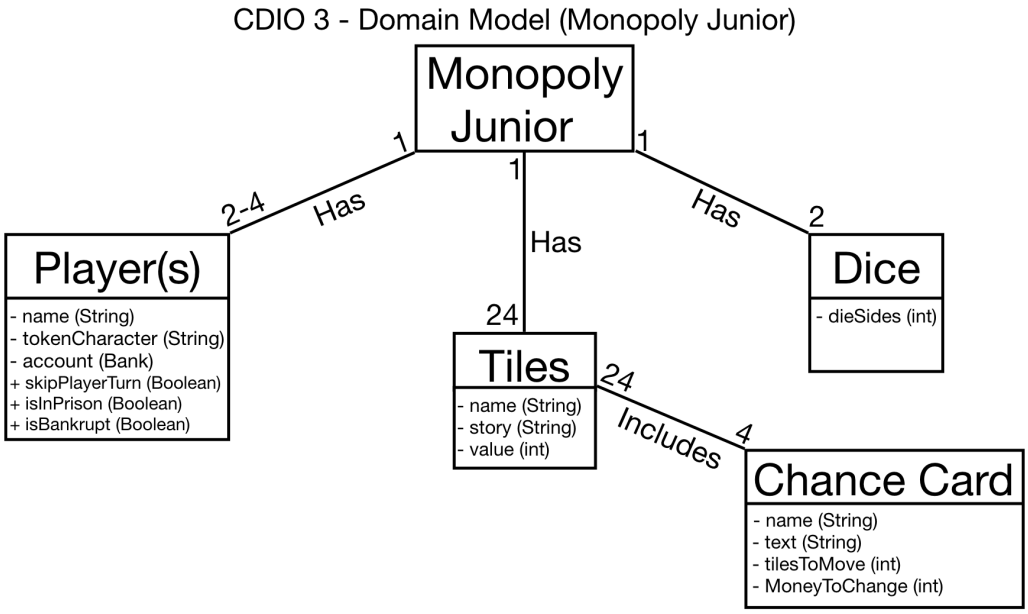
Design Sequence diagram

The group did not only choose to use the system sequence diagram, but has also used the design sequence diagram. Instead of showing how the player interact with the game, it shoes how the game interact with player, so the other way around. This mean it shows what the game it-self does when the player does something.



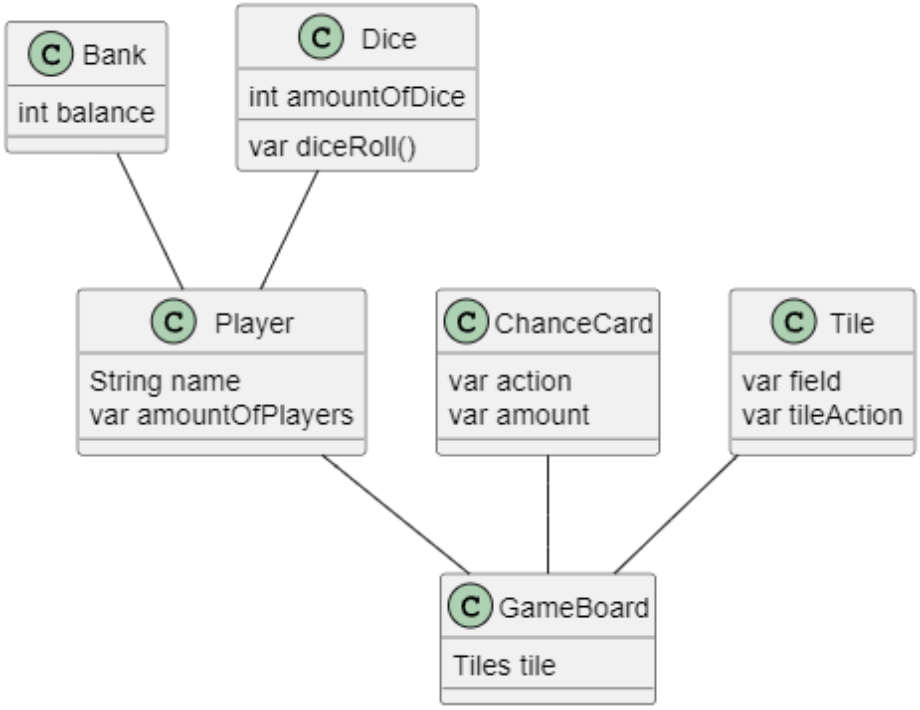
Domain model

The domain model is used to show all the classes, what each class contains and the relationships between them. All the boxes is a class, with the giving name of that class. In the classes you can see all the methods and if they are either private or public. The lines between the classes show which classes interact with each other. On the lines which indicates the relationships, there is a description of their relationship.


CDIO 3 - Domain Model (Monopoly Junior)
Monopoly Junior
1
1
1
Has
2-4
Has
Has
2
24
Player(s)
- name (String)
- tokenCharacter (String)
- account (Bank)
+ skipPlayerTurn (Boolean)
+ isInPrison (Boolean)
+ isBankrupt (Boolean)
Dice
- dieSides (int)
Tiles
- name (String)
- story (String)
- value (int)
24
Includes
4
Chance Card
- name (String)
- text (String)
- tilesToMove (int)
- MoneyToChange (int)

Class Diagram

The class diagram is used to show how the different classes work together. It is used to show what methods and variables the different classes have, as well as what classes specifically work together.



In our case this class diagram was our first iteration, to give us a basic idea of what classes we needed and some basic ideas as to what methods and variables each class should be responsible for. Since then we have added other classes such as the "Field" class that is now responsible for the different type of fields. And we also noticed that we had a lot more methods to incorporate in the game, so this class diagram is per definition outdated. The reason we keep it in the report, is to show the difference in our initial idea of the game and our final product. This is also to show our agile approach to the development process, where we can go back and change different aspects of the program accordingly.

6 Implementation

The group has had many thoughts and work doing the project, before starting programming the game itself. To find out how we could make the best possible program and at the same time meet the requirements for the program. The group has therefore used FURPS ("functionality", "usability", "reliability", "performance" and "supportability").

FURPS

The group have as said used FURPS for the project. FURPS stands for "functionality","usability", "reliability", "performance" and "supportability". This has been used to classify the quality of the software, which means the program.

Functionality

The functionality of the program is very simple. It is not a game with many requirements and rules. The group also chose not to have all the rules from the original Monopoly Junior game. This is done because it's always beneficial to have a program with a very simple functionality, it makes the game more user friendly. Some of the rules were also just not fit for a terminal based game.

Usability

The usability of the program is good. To improve usability for the Monopoly Junior game, the group has created a .bat file that launches the game for the user. Which means it only takes a few clicks to open and start the game for the user. Therefore the usability could almost not be any better.

Reliability

The reliability of the program is great. The software should work almost perfectly and the chance of a program crash is almost zero. It's a small program and therefore it doesn't take much for the program to run and work.

Performance

The performance of the program is good. The program requires almost nothing to run. Once again because it's a small program with few requirements. The user will be able to run the program on any modern computer no matter the specifications of the device. The only thing is that the user have to run the game on java 13 or a newer version, furthermore the computer should also support running a ".bat" file, which compiles everything needed for the user. The game can also run on the computers in "DTU's databarer". The response time for the program is very good as well.

Supportability

The supportability of the program is good. Since it is a small program. Therefore it is simple to test, adjust, adapt, install, maintain, configure, and localize.

GRASP

GRASP is a collection of principles in software development. It revolves around object design and management of responsibilities. GRASP is an abbreviation of **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns. These aforementioned patterns are controller, creator, indirection, information expert, low coupling, high cohesion, polymorphism, protected variations and pure fabrication. In the following segments we will discuss how we have used some of these principles in our work and how some of them have not been used as much as they should.

Polymorphism

We have used polymorphism in our class design of our different fields. Our super-class, field, has only attributes that all fields also make use of. All sub-classes that extend the field class, then use the constructor of the super-class, using super(name). This is not polymorphism, but we do use polymorphism in terms of the getDescription() method. Here each class overrides the method in the superclass to utilise polymorphism. This results in each subclass having its own method for getDescription, which is useful as each subclass of field is

different.

Low coupling and High cohesion

When we began the development of the game we were very keen to use low coupling and high cohesion. We also kept this in mind while developing the software, but as the game grew and we had more classes that needed to work together, the low coupling started to get less used. This can be seen as we in many different classes use methods from the same class e.g the player class. We feel that although low coupling hasn't turned out great, our cohesion is a bit better. Each class has a very specific job and tries to do that as best as possible, even though it is dependent on other classes.

6.1 Testing

Our first test case is in regards to a player's bank balance. One of the rules in the game is that the game ends when a player goes bankrupt, therefore it is essential that a player can't have a negative bank balance. To test this we have created a JUnit test, which checks some of our methods. At first a new player is created with a bank balance of 20. Our getBankBalance method is then called, and we expect it returns 20, which it did. The next test in our JUnit test, calls the setBankBalance method that serves the purpose of changing a players bank balance. First we add 2 to the players account and then we withdraw 25 from the players account, which should take the balance below, if there are faults in our code. But the test passed as the getBankBalance method returned 0. At last we test our toString method to see if it returns the correct string with the players number and bank balance. With this JUnit test we can conclude that the method tested all work as intended.

The second test case is all about what happens if the users input does not match the required input. We have created the following equivalence class test to ensure a program behaves as expected.

Equivalence class test

Input	Expected output	Actual output
-1	"Received an invalid amount"	"Received an invalid amount"
0; 1	"Received an invalid amount"	"Received an invalid amount"
2; 3; 4;	The game works as intended	The game works as intended
5; 10	"Received an invalid amount"	"Received an invalid amount"
"four"	"didn't receive an integer"	"didn't receive an integer"
""	"didn't receive an integer"	"didn't receive an integer"

In this test we have created six equivalence classes, the first being negative numbers, the second border value and 0, the third valid values, the fourth just outside the upper border and a two digit number, the fifth being a string, and the sixth being the empty input. When we performed the test we realised nothing happened when the input was empty, and then

we went and addressed that issue. Our program now passes the test.

The third test case we wanted to test was with a user. As the user we used one of the group members mother who has no experience with software development. We sat her down with the program and didn't tell her anything about how it works, other than it was a monopoly junior game. It went well she didn't have any issues, so we decided to make her input invalid inputs when choosing the amount of players. She had no issues here either, but she pointed out that she didn't know what an integer when the program told her to "enter an integer between 2 and 4". We could see her point so we decided to change it to say "enter a number between 2 and 4". After this test, and the correction made, we a satisfied with how user friendly our program is.

7 Conclusion

We have created a monopoly junior program, which runs in the terminal and can be played by two, three or four players. The program has been developed with object oriented principles and multiple classes that have various responsibilities. Github was used in the development process, so all members of the group had access too all the files at all times, and giving us the ability to have version control on the program. We have used UML diagrams to design our program and give us a better understanding of how all the files and classes work and is supposed to work together. In regards to testing we have primarily work with three test cases. The first being a JUnit test which ensures that a players bank balance can't go below zero. The second was an equivalence test which checks what happens when various inputs a given to the program, and the third and final test case was an user that served the purpose of finding out how user friendly the program was. All the methods are used together to create a user friendly and stable monopoly junior game.

8 Appendix

8.1 Litterature

1. CDIO1 made by Group 07 [CDIO1-G7]
2. CDIO2 made by Group 07 [CDIO2-G7]
3. GitHub repository: [`https://github.com/adel051700/G07_CDIO3`]

8.2 Our code

```
class Player {
    private int number;
    private Bank account;
    public boolean skipPlayerTurn = false;
    public boolean isInPrison = false;
    private boolean getOutOfJailFreeCard = false;
    private int position;

    public Player (int number, int bankBalance) {
        this.number = number;
        this.account = new Bank();
        this.account.changeBalance(bankBalance);
        this.position = 0;
    }
    public boolean getOutOfJailFreeCard()
    {
        return this.getOutOfJailFreeCard;
    }
    public void changeOutOfJailFreeCard()
    {
        this.getOutOfJailFreeCard = !this.getOutOfJailFreeCard;
    }

    public int getNumber()
    {
        return this.number;
    }

    public Boolean getSkipPlayerTurn() {
        return skipPlayerTurn;
    }

    public void setSkipPlayerTurn() {
        this.skipPlayerTurn = !getSkipPlayerTurn();
    }

    public Boolean getIsInPrison() {
        return isInPrison;
    }

    public void changeIsInPrison() {
        this.isInPrison = !getIsInPrison();
    }

    public String toString() {
        return "Player " + this.number + System.lineSeparator() + "Bank balance: " + this.account.getBalance();
    }

    public int getBankBalance() {
        return account.getBalance();
    }

    public void setBankBalance(int alphaChange)
    {
        account.changeBalance(alphaChange);
    }
    public void setPosition(int tileNumber)
    {

        this.position += tileNumber;
        if (this.position >= 24)
        {
            System.out.println("You passed Start and recieve 2$");
            this.setBankBalance(2);
            this.position %= 24;
        }
    }
    public int getPosition()
    {
        return this.position;
    }
}

```

Figure 1: The player class creates a player with a player number, an account from the bank class, the option to change the amount in the account, a position and if they have a 'get out of jail free' card. The class has many get and set methods for the different values the player has

```java
class Field {
    public String name;
    private Player owner = null;

    public Field(String name) {
        this.name = name;
    }
    public Player getOwner()
    {
        return this.owner;
    }
    public int getValue()
    {
        return 0;
    }
    public String getColor() {
        return "";
    }
    public String getName()
    {
        return this.name;
    }
    public void setMultiplier(int fillerFunctionForFunctionalityInBuyableFields)
    {}


    public String getDescription(Player player, Player[] playerArr, Field[] gameBoard) {
        String returnStatement = "You have landed on " + this.name + ", nothing further happens...";
        return returnStatement;
    }
}
```

Figure 2: The "field" class is a super class, that creates an object, field, with a name. The super class provides methods to get the name and the description of the field. Everything value beside the name of the object to an empty string or 0, because the subclasses have different values.

```java
class buyableField extends Field {
    private int value;
    private String color;
    private int multiplier;
    private Player owner;

    public buyableField(String name, int value, String color, int multiplier, Player owner) {
        super(name);
        this.value = value;
        this.color = color;
        this.multiplier = multiplier;
        this.owner = owner;
    }
    @Override
    public Player getOwner()
    {
        return this.owner;
    }
    @Override
    public int getValue()
    {
        return this.value;
    }
    @Override
    public String getColor() {
        return this.color;
    }

    public void setMultiplier(int multiplier) {
        this.multiplier = multiplier;
    }
    @Override
    public String getDescription(Player player, Player[] playerArr, Field[] gameBoard, boolean getFree) {
        String returnStatement = "You landed on " + this.name + " which is a " + this.color + " tile";
        if(this.owner == null && getFree && owner != player) {
            this.owner = player;
            returnStatement += "\n This tile isnt owned by anyone, so you get it for free!";
        }
        if (owner != null && !owner.equals(player))
        {
            returnStatement+= "\n The tile is owned by: player number " + this.owner.getNumber();
            returnStatement += "\n You pay " + this.value + "$ in rent to: player number " + this.owner.getNumber() + ".";
            this.owner.setBankBalance(value*multiplier);
            player.setBankBalance(-value*multiplier);
        }
        else if (owner == player)
        {
            returnStatement += "\n You already own this tile, and nothing further happens...";
        }
        else
        {

            returnStatement += "\n You buy this tile for " + this.value + "$";
            player.setBankBalance(-value);
            this.owner = player;
        }

        returnStatement += "\n You now have " + player.getBankBalance() + "$ left";


        return returnStatement;
    }
}
```

Figure 3: The buyable Field is a subclass of the superclass "Field", when this sub field is initialized, the subclass gives the object, field, a name through super(name), it also gives the object a value, a color, a multiplier value, and an owner, through a constructor. The subclass provides get methods for the owner, value, color, it also provides a set method for the multiplier. Lastly it provides a get method for the description, where it checks if the player that land on the field, get the field for free through a chance card, if the field is owned by another player, so the current player has to pay rent or either if the field is either owned by the current player or the field is ownerless, and has to be bought by the player. Buyable field isn't the only subclass of "field", there is also the "specialField", "chanceField", "prisonField". the "specialField" include the tiles, Go, Visting jail and free parking, which only provides a name and getdescription() method. The "chanceField" include all the tiles in which the player has to draw a chance card, it provides the name of the field, a method to draw a chance card and a getdescription() method. Lastly the "prisonField", which only includes the 'Go to jail' tile, and provides the name and getdescription() method.

```
// Creates array of players with given length;
        Player[] players = new Player[n];



        for (int i = 0; i < players.length; i++)
        {
            players[i] = new Player(i+1,(24-(n*2)));
        }
        int playerTurn = 0;
        Player activePlayer;
        boolean loseCondition = false;
        //
        //
        // Main game loop:
        //
        //
        while (!loseCondition)
        {
            System.out.println(Gameboard.toString(gameBoard,players));
            System.out.println("\n");

            playerTurn %= n;
            System.out.println("It is player number " +(playerTurn+1)+" turn, press enter to roll the dice:");
            s.nextLine();
            activePlayer = players[playerTurn];

            if (activePlayer.getSkipPlayerTurn())
            {
                System.out.println("Your turn is skipped.");
                playerTurn++;
                continue;
            }

            if (activePlayer.getIsInPrison())
            {
                System.out.println("You are in prison, and must therefore pay 1$ to continue playing");
                activePlayer.setBankBalance(-1);
                System.out.println("Your balance is now, " + activePlayer.getBankBalance()+ "$");
                if (activePlayer.getBankBalance()==0)
                {
                    break;
                }
                else
                {
                    activePlayer.changeIsInPrison();
                    playerTurn++;
                }
            }

            die1 = dice.roll();
            die2 = dice.roll();
            //System.out.println(activePlayer.getPosition());
            //System.out.println(die1 + " " + die2);
            activePlayer.setPosition(activePlayer.getPosition() + die1 + die2);
            System.out.println(die1 + " " + die2);
            System.out.println(gameBoard[activePlayer.getPosition()].getDescription(activePlayer,players,gameBoard,false));


            if (die1 != die2)
            {
                playerTurn++;
            }
            else
            {
                System.out.println("You rolled two " + die1 + "\n It is therefore your turn again");
            }

            for (int k = 0; k < players.length; k++)
            {
                if (players[k].getBankBalance() == 0)
                {
                    // Breaks to calculate the winner(s);
                    loseCondition = true;
                }
            }
            for (int j = 0; j < gameBoard.length-1; j++)
            {
                if (gameBoard[j].getOwner()!=null && gameBoard[j+1].getOwner() != null)
                {
                    if (gameBoard[j].getOwner().equals(gameBoard[j+1].getOwner()))
                    {
                        gameBoard[j].setMultiplier(2);
                    }
                }
            }

        }
        Player maxVal = new Player(0,0);
        Player minVal = new Player(0,0);
        for (int i = 0; i < players.length; i++)
        {
            if (players[i].getBankBalance() > maxVal.getBankBalance())
            {
                maxVal = players[i];
            }
            if (players[i].getBankBalance()==0)
            {
                minVal = players[i];
            }
        }
        System.out.println("player number: " + minVal.getNumber() + "has gone bankrupt");

        System.out.println("The Winner is player number: " + maxVal.getNumber() + "\n with a score of " + maxVal.getBankBalance());

        }
        catch(IllegalArgumentException e)
        {
            System.out.println(e.getMessage());
            //s.next();
            continue;
        }
        s.close();
        break;
    }
```

Figure 4: This is the game loop of the monoply game, it is located in the monopoly.java file along side exceptions to ensure the game starts correctly with the correct amount of player. The games makes use of the classes: Dice, field, Gameboard and Player, to make sure the game runs correctly